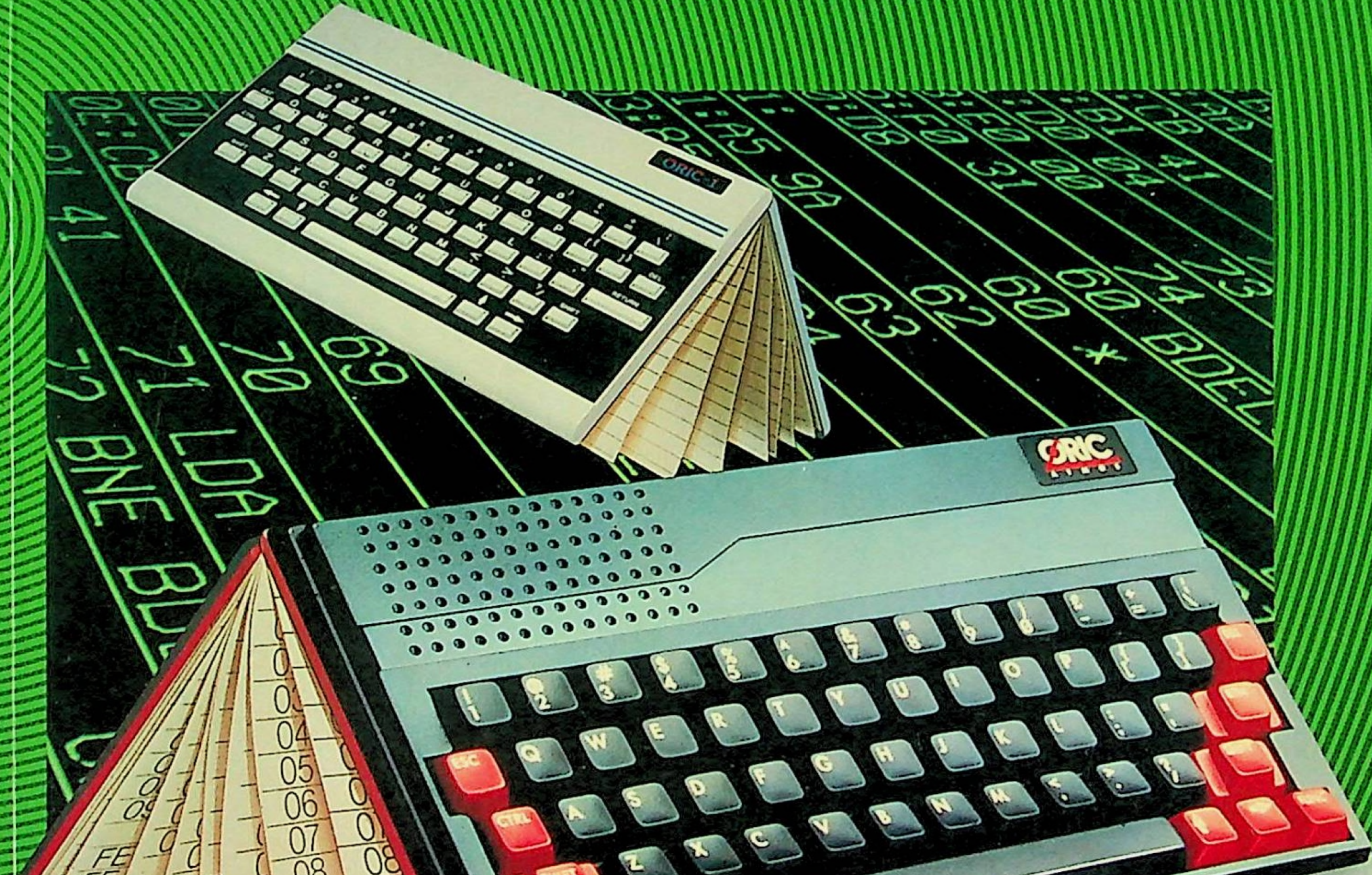
Machine Code for the Atmos and Oric-1

SHIVA's friendly micro series

Bruce Smith



Machine Code for the Atmos and Oric-1

Bruce Smith

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
4 Church Lane, Nantwich, Cheshire CW5 5RQ, England

ISBN 0 906812 69 0

Cover photographs of the Atmos and Oric-1 courtesy of Oric Products International Ltd.

Typeset by Gilbert Composing Services, Leighton Buzzard
and printed by Devon Print Group, Exeter.

Contents

Preface

At the very heart of your Oric Atmos or Oric-1 microcomputer is an inconspicuous chip called the 6502 microprocessor. It is responsible for co-ordinating every single thing your Oric does all the time it is switched on—no mean feat! However, no matter what preconceptions you may have, programming the Oric at its own machine code level is not difficult—and that is the very aim of this book—to teach you just how to develop and write your own machine code programs.

The text assumes that you have some knowledge of Oric BASIC but know absolutely nothing about machine code, though you will probably have read the machine code chapter in the *Oric Atmos Manual or Oric-1 BASIC Programming Manual.* I have tried very hard to write in non-technical language and set the chapters out in a logical manner, introducing new concepts in digestible pieces as and when they are needed rather than devoting chapters to specific items. Wherever possible, practical programs are included to bring home the point being made, and in most instances these are analysed and the function and operation of each instruction explained.

Machine Code for the Atmos and Oric-1 is completely self-contained and includes a full description of all the machine code instructions available and suggests suitable applications for their use. After a 'bit of theory' in the opening chapters, the main registers of the 6502 are introduced and descriptions given of how, when and where machine code routines can be entered. There is also a simple machine code monitor program which facilitates the entry of such routines.

After discussing the way in which the 6502 flags certain conditions to the outside world, some of the modes of addressing the chip are described. Machine code addition and subtraction are introduced and the easiest ways of manipulating and saving data for future use by the program and processor are described. Machine code loops (equivalent to BASIC's FOR . . . NEXT . . . STEP . . .) show how sections of code may be repeated, and subroutines and jumps take the place of BASIC's GOSUB and GOTO. Also included is a look at some more complicated procedures, such as multiplication and division using the shift and rotate instructions.

Finally, the Appendices provide a quick and easy reference to the sorts of things you'll need to 'want to know quickly' when you start writing your very own original machine code programs!

Highbury, November 1983 Bruce Smith

1 Machine Code or Assembly Language?

The 6502 microprocessor within your Oric microcomputer can perform 152 different operations, with each one being defined by a number (or *operation code*) in the range 0 to 255 (see Appendix 5). To create a machine code program we need simply to POKE successive memory locations with the relevant operation codes—'opcodes' for short. For example, to store the value 5 into location #1500 (in other words to do the machine code equivalent of BASIC's POKE #1500, 5), we would need to POKE the following bytes into memory:

```
#A9
#05
#8D
#00
#15
```

and then ask the Oric's 6502 to execute them.

Not exactly clear is it! That's where *assembly language* comes in.

Assembly language allows us to write machine code in an abbreviated form which is designed to represent the actual operation the opcode will perform. This abbreviated form is known as a *mnemonic* and it is the basic building block of assembly language (or assembler) programs.

We could rewrite the previous machine code in assembler like this:

```
LDA @5
STA #1500
```

and it can be read as:

Load the **a**ccumulator with the value 5

Store the **a**ccumulator's contents at location #1500

As you can see from the **bold** letters, the mnemonic is composed of letters in the instruction, which greatly enhances its readability.

Once the assembler program is complete, it can be converted into machine code in one of two ways.

1. With the aid of a *mnemonic assembler.* This is itself a program (written in machine code or BASIC) which transforms the assembly language instructions (known as the source) into machine code (known as the object code) and POKEs them into memory as it does so. Commercial versions will no doubt be forthcoming as the Oric increases in popularity.

2. By looking up the relative codes in a table and then POKEing them into memory using a monitor program or a DATA-reading FOR . . . NEXT loop. Full details of this method are given in Chapter 5, which also includes a simple monitor program.

All the programs in this book are listed in both their assembler *and* machine code forms, so they can be entered by either of the above methods.

Appendix 3 provides comprehensive user information about *all* of the 65Ø2's opcodes, so don't worry too much if some of this seems a bit foreign at the moment—we'll soon change that!

WHY MACHINE CODE?

Why bother to talk to the Oric in its own language? There are really two main reasons. Firstly speed. Machine code is executed very much faster than a high level language such as BASIC. Remember that the BASIC interpreter is itself written in machine code and the BASIC statements and commands are simply pointers to routines within the BASIC ROM. However, because each statement and command must be interpreted and located within the ROM, a decrease in operation speed occurs. Secondly, learning machine code allows you to understand just how your computer works, and lets you create special effects and routines not possible within the constraints imposed by the limited set of BASIC instructions. Machine code allows you to control your Oric, rather than it controlling you!

2 Numbers

BINARY, HEX AND DECIMAL

We have seen that the instructions the Oric operates with consist of sequences of numbers. But just how are these numbers stored internally? Well, not wishing to baffle you with the wonders of modern computer science, let's try to simplify matters somewhat and say that each instruction is stored internally as a binary number. Decimal numbers are composed of combinations of ten different digits, that is 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 and are said to work to a *base* of 10. As its name suggests, binary numbers work to a base of 2 where only the digits 0 and 1 are available. These two numbers represent the two different electrical conditions that are available inside the Oric, namely 0 volts (off) and 5 volts (on).

The machine code already described is therefore represented internally as:

Mnemonic	Machine code	Binary
LDA	A9	10101001
@#5	05	00000101
STA	8D	10001101
00	00	00000000
#15	15	00010101

As can be seen, each machine code instruction is expressed as eight binary digits, called *bits*, which are collectively termed a *byte*.

Usually each of the bits in a byte is numbered for convenience as follows:

7	6	5	4	3	2	1	0

The number of the bit increases from right to left, but this is not so odd as it may first seem.

Consider the decimal number 2934, we read this as two thousand, nine hundred and thirty four. The highest numerical value, two thousand, is on the left, whilst the lowest, four, is on the right. We can see from this that the position of the digit in the number is very important, as it will affect its *weight*.

The second row of Table 2.1 introduces a new numerical representation. Each base value is postfixed with a small number or *power*, which corresponds to its overall position in the number. Thus 10^3, read as ten raised to the power of three, simply implies $10 \times 10 \times 10 = 1000$.

Table 2.1

Value	1000s	100s	10s	1s
Representation	10^3	10^2	10^1	10^0
Digit	2	9	3	4

In binary representation, the weight of each bit is calculated by raising the base value, two, to the bit position (see Table 2.2). For example bit number 7 has a notational representation of 2^7 which expands to: $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 128$!

Table 2.2

Bit number	7	6	5	4	3	2	1	0
Representation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight	128	64	32	16	8	4	2	1

BINARY TO DECIMAL CONVERSION

As it is possible to calculate the weight of individual bits, it is a simple matter to convert binary numbers into decimal numbers. The rules for conversion are:

If the bit is *set*—that is it contains a 1—add its weight
If the bit is *clear*—that is it contains a 0—ignore its weight

Let us try an example and convert the binary number 10101010 into its equivalent decimal value.

$$\begin{aligned}
1 \times 128(2^7) &= 128\\
0 \times 64(2^6) &= 0\\
1 \times 32(2^5) &= 32\\
0 \times 16(2^4) &= 0\\
1 \times 8(2^3) &= 8\\
0 \times 4(2^2) &= 0\\
1 \times 2(2^1) &= 2\\
0 \times 1(2^0) &= 0\\
\hline
&\ 170
\end{aligned}$$

Therefore 10101010 binary is 170 decimal.

Similarly 11101110 represents:

$$\begin{array}{rrrl}
1 \times & 128(2^7) & = & 128 \\
1 \times & 64(2^6) & = & 64 \\
1 \times & 32(2^5) & = & 32 \\
0 \times & 16(2^4) & = & 0 \\
1 \times & 8(2^3) & = & 8 \\
1 \times & 4(2^2) & = & 4 \\
1 \times & 2(2^1) & = & 2 \\
0 \times & 1(2^0) & = & 0 \\
\hline
 & & & 238
\end{array}$$

in decimal.

DECIMAL TO BINARY CONVERSION

To convert a decimal number into a binary one, the procedure described earlier is reversed—each binary weight is subtracted in turn. If the subtraction is possible, a 1 is placed into the binary column and the remainder carried down to the next row where the next binary weight is subtracted.

If the subtraction is not possible, a 0 is placed in the binary column and the number moved down to the next row. For example, the decimal number 141 is converted into binary as in Table 2.3.

Table 2.3

Decimal number	Binary weight	Binary	Remainder
141	128(2^7)	1	13
13	64(2^6)	0	13
13	32(2^5)	0	13
13	16(2^4)	0	13
13	8(2^3)	1	5
5	4(2^2)	1	1
1	2(2^1)	0	1
1	1(2^0)	1	0

Therefore 141 = 10001101 binary.

BINARY TO HEX CONVERSION

Although binary notation is probably as close as we can come to representing the way numbers are stored within the Oric, you will no doubt have noticed that the machine code examples consist of a series of two characters preceded by a hash, '#'. This type of number is known as a *hexadecimal* number, or hex for short, and its value is calculated to a base of 16! This, at first sight, may seem singularly awkward, however it does present several distinct advantages over binary and decimal numbers as we shall see.

Sixteen different characters are required to represent all the possible digits in a hex number. To produce these, the numbers 0 to 9 are retained, and the letters A, B, C, D, E and F are used to denote the values 10 to 15. Binary conversion values are shown in Table 2.4.

Table 2.4

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

To convert a binary number into hex, the byte must be separated into two sets of four bits, termed *nibbles*, and the corresponding hex value of each nibble extracted from Table 2.4.

Example Convert 0110 1001 to hex:

$$\begin{tikzcd}
\text{High nibble} \arrow[dr] & & & & \text{Low nibble} \arrow[dl] \\
& 0110 \arrow[dr] & & 1001 \arrow[dl] & \\
& & 69 & &
\end{tikzcd}$$

Because it is not always apparent whether a number is hex or decimal (as in the example above), hex numbers in the Oric are always preceded by a hash—therefore 01101001 is #69 (read hex six nine).

By reversing the process, hex numbers can readily be converted into binary.

Example Convert #AF to binary:

$$\begin{tikzcd}
& AF \arrow[dl] \arrow[dr] & \\
1010 \arrow[dr] & & 1111 \arrow[dl] \\
& 10101111 &
\end{tikzcd}$$

It should now be apparent that hex numbers are much easier to convert to binary (and vice versa), than their decimal counterparts, and the maximum binary number possible with one byte, 11111111, requires just two hex digits, #FF.

HEX TO DECIMAL CONVERSION

For the sake of completeness, let's see how hex and decimal numbers may be converted. To transform a hex number into decimal, the decimal weight of each digit should be summed.

Example convert #31A to decimal:

The 3 has the value $3 \times 16^2 = 3 \times 16 \times 16 = 768$

The 1 has the value $1 \times 16^1 = 1 \times 16 = 16$

The A has the value $1 \times 16^0 = 10 \times 1 = 10$

add these together to give #31A = 794 decimal.

Converting decimal to hex is a bit more involved and requires the number to be repeatedly divided by 16 until a value less than 16 is obtained. This hex value is noted, and the remainder carried forward for further division. This process is continued until the remainder itself is less than 16.

Example convert 4072 to hex:

$4072 \div 16 \div 16 = 15 = F$ (remainder $= 4072 - (15 \times 16 \times 16) = 232$)

$232 \div 16 = 14 = E$ (remainder $= 232 - (14 \times 16) = 8$)

$8 = 8$

Therefore 4072 decimal is #FE8.

Both of these conversions are a little long winded (to say the least!) and after all we do have a very sophisticated microcomputer available to us, so let's make it do some of this more tedious work!

To print the decimal value of a hex number, such as #31A enter:

```
PRINT #31A
```

and to print the hex value of a decimal number use:

```
PRINT HEX$(4072)
```

3 Logically it Adds Up!

BINARY ARITHMETIC

Please don't be put off and skip this chapter simply because it contains that dreaded word—arithmetic. The addition and subtraction of binary numbers is simple, in fact if you can count to two you will have no problems whatsoever! Although it is not vital to be able to add and subtract ones and noughts by 'hand', this chapter will introduce several new concepts which are important, and will help you in your understanding of the next few chapters.

ADDITION

There are just four, simple, straightforward rules when it comes to adding binary numbers. They are:

1. Ø + Ø = Ø
2. 1 + Ø = 1
3. Ø + 1 = 1
4. 1 + 1 = (1)Ø

Note, that in rule 4, the result of 1 + 1 is (1)Ø. The 1 in brackets is called a *carry* bit, and its function is to denote an overflow from one column to another, remember, 1Ø binary is 2 decimal. The binary 'carry' bit is quite similar to the carry that can occur when adding two decimal numbers together whose result is greater than 9. For example, adding together 9 + 1 we obtain a result of 1Ø (ten), this was obtained by placing a zero in the units column and carrying the 'overflow' across to the next column to give: 9 + 1 = 1Ø. Similarly, in binary addition when the result is greater than 1, we take the carry bit across to add to the next column.

Let's try to apply these principles to add together two 4 bit binary numbers, Ø1Ø1 and Ø1ØØ.

```
   Ø1Ø1          (#5)
 + Ø1ØØ          (#4)
   ----
   1ØØ1          (#9)
```

Reading each individual column from right to left:

```
First column:    1 + Ø             = 1
Second column:   Ø + Ø             = Ø
Third column:    1 + 1             = Ø (1)
Fourth column:   Ø + Ø = Ø + (1)   = 1
```

In this example a carry bit was generated in the third column, and was carried across and added to the fourth column.

Adding 8 bit numbers is accomplished in a similar manner:

```
   01010101        (#55)
 + 01110010        (#72)
   --------
   11000111        (#C7)
```

SUBTRACTION

So far we have been dealing with positive numbers, however in the subtraction of binary numbers we need to be able to represent negative numbers as well as positive ones. In binary subtraction though, a slightly different technique from normal everyday subtraction is used, in fact we don't really perform a subtraction at all—we add the negative value of the number to be subtracted. For example, instead of executing 4 − 3 (four minus three) we actually execute 4 + (−3) (four, plus minus three)! Figure 3.1 will hopefully eradicate any confusion or headaches that may be prevailing!

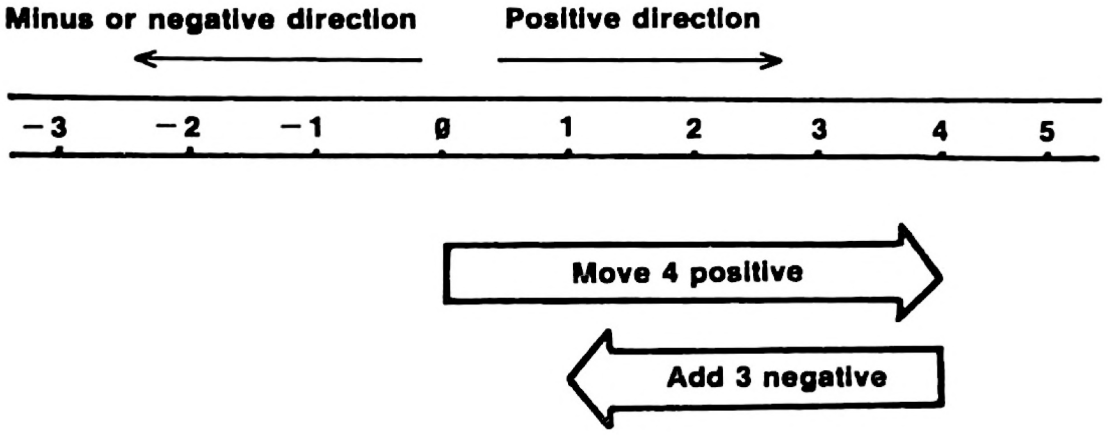


Figure 3.1 Diagramatic representation of 4 + (− 3).

We can use the scale to perform the example 4 + (−3). The starting point is zero. First move to point 4 (i.e. four points in a positive direction) and *add* to this −3 (i.e. move three points in a negative direction). We are now positioned at point 1 which is, of course, where we should be. Try using this method to subtract 8 from 12, to get the principle clear in your mind.

Okay, lets now see how we apply this to binary numbers, but first, just how are negative numbers represented in binary? Well, a system known as *signed binary* is employed, where bit 7, known as the most significant bit (msb), is used to denote the *sign* of the number. Traditionally a '0' in bit 7 denotes a positive number and a '1' a negative number. For instance, in signed binary:

```
1 0000001
↑   └──────Bits 0–6 give value = 1
└──────────Sign bit = 1, therefore number is negative
```

so, 10000001 = −1. And:

```
0 1111111
↑   └──────Bits 0–6 give value = 127
└──────────Sign = 0 therefore number is positive
```

therefore 01111111 = 127.

However, just adjusting the value of bit 7 as required, is not an accurate way of representing negative numbers. What we must do to convert a number into its negative counterpart, is to obtain its *two's complement* value. To do this simply invert each bit and then add one.

To represent −3 in binary, first write the binary for 3:

```
0 0 0 0 0 0 1 1
```

Now invert each bit. (Replace each 0 with a 1, and each 1 with a 0—this is known as its *one's complement.*

```
1 1 1 1 1 1 0 0
```

Now add 1:

```
      1 1 1 1 1 1 0 0
 +                  1
      ---------------
      1 1 1 1 1 1 0 1
```

Thus, the two's complement value of −3 = 11111101. Let us now apply this to our original sum 4 + (−3):

```
(4)                0 0 0 0 0 1 0 0
(−3)               1 1 1 1 1 1 0 1
                   ---------------
(Now add)       (1)0 0 0 0 0 0 0 1
```

We can see that the result is 1 as we would expect, but we have also generated a carry bit due to an overflow from bit 7. This carry bit can be ignored for our purposes at present, though it does have a certain importance as we shall see later on.

A further example may be of use. Perform 32 − 16 i.e. 32 + (−16).

32 in binary is:

```
0 0 1 0 0 0 0 0
```

16 in binary is:

```
0 0 0 1 0 0 0 0
```

The two's complement of 16 is:

```
      1 1 1 0 1 1 1 1
 +                  1
      ---------------
      1 1 1 1 0 0 0 0
```

Now add the two together:

```
(32)                0 0 1 0 0 0 0 0
(−16)             + 1 1 1 1 0 0 0 0
                    ---------------
(16)            (1) 0 0 0 1 0 0 0 0
```

Ignoring the carry, we have our result, 16.

We can see from these examples that, using the rules of binary addition, it is possible to add or subtract signed numbers. If the 'carry' is ignored, the result, including the sign, is correct. Thus it is also possible to add two negative values together and still obtain a correct negative result. Using two's complement signed binary let's perform (−2) + (−2).

2 in binary is:

```
    0 0 0 0 0 0 1 0
```

The two's complement value is:

```
       1 1 1 1 1 1 0 1
    +                1
       ---------------
       1 1 1 1 1 1 1 0
```

We can add this value twice to perform the addition:

```
    (-2)           1 1 1 1 1 1 1 0
    (-2)        +  1 1 1 1 1 1 1 0
                   ---------------
                (1) 1 1 1 1 1 1 0 0
```

Ignoring the carry, the final result is −4. You might like to confirm this by obtaining the two's complement value of −4 in the usual manner.

LOGICAL OPERATIONS

The theory of logic is based on situations where there can only ever be two possibilities, namely *yes* and *no*. In binary terms these two possibilities are represented as 1 and 0.

There are three different logical operations that can be performed on binary numbers, they are AND, OR and EOR. In each case the logical operation is performed between the corresponding bits of two separate numbers.

AND

The four rules for AND are:

1. 0 AND 0 = 0
2. 1 AND 0 = 0
3. 0 AND 1 = 0
4. 1 AND 1 = 1

As can clearly be seen, the AND operation will only generate a 1 if *both* of the corresponding bits being tested are 1. If a 0 exists in either of the corresponding bits being tested, the resulting bit will always be 0.

Example AND the following two binary numbers:

```
         1 0 1 0
    AND  0 0 1 1
         -------
         0 0 1 0
```

In the result only bit 1 is set, the other bits are all clear because in each case one of the bits being tested contains a 0.

The main use of the AND operation is to 'mask' or 'preserve' certain bits. Imagine that we wish to preserve the low four bits of a byte (low nibble) and completely clear the high four bits (high nibble). We would need to AND the number with 00001111. If the other byte contained 10101100 the result would be given by:

```
        1 0 1 0 1 1 0 0         (byte being tested)
   AND  0 0 0 0 1 1 1 1         (mask)
        ---------------
        0 0 0 0 1 1 0 0
```

the high nibble is cleared and the low nibble preserved!

OR

The four rules for OR are:

1. 0 OR 0 = 0
2. 1 OR 0 = 1
3. 0 OR 1 = 1
4. 1 OR 1 = 1

Here the OR operation will result in a 1 if *either* or *both* the bits contain a 1. A 0 will only occur if *neither* of the bits contains a 1.

Example OR the following two binary numbers:

```
        1 0 1 0
   OR   0 0 1 1
        -------
        1 0 1 1
```

Here, only bit 2 is clear, the other bits are all set as each pair of tested bits contains at least one 1.

One common use of the OR operation is to ensure that a certain bit (or bits) is set—this is sometimes called 'forcing bits'. As an example, if you wish to force bit 0 and bit 7, you would need to OR the other byte with 10000001.

```
        0 0 1 1 0 1 1 0         (byte being tested)
   OR   1 0 0 0 0 0 0 1         (forcing byte)
        ---------------
        1 0 1 1 0 1 1 1
```

The initial bits are preserved, but bit 0 and bit 7 are 'forced' to 1.

EOR

Like AND and OR, this donkey sounding operation has four rules:

1. 0 EOR 0 = 0
2. 1 EOR 0 = 1
3. 0 EOR 1 = 1
4. 1 EOR 1 = 0

This operation is exclusive to OR, in other words, if both bits being tested are similar a 0 will result. A 1 will only be generated if the corresponding bits are unlike.

Example EOR the following two binary numbers:

```
        1 0 1 0
EOR     0 0 1 1
        -------
        1 0 0 1
```

This instruction is often used to complement, or invert, a number. Do this by EORing the other byte with 11111111.

```
        1 0 0 1 1 0 0 0          (byte being inverted)
EOR     1 1 1 1 1 1 1 1          (inverting byte)
        ---------------
        0 1 1 0 0 1 1 1
```

Compare the result with the first byte, it is completely opposite.

4 The Registers

To enable the 6502 to carry out its various operations, it contains within it several special locations, called *registers*. Because these registers are internal to the 6502, they do not appear as part of the Oric's memory map (see Appendix 6), and are therefore referred to by name only. Figure 4.1 shows the typical programming model of the 6502. For the time being we need only concern ourselves with the first four of these six registers, they are the accumulator, the X and Y registers and the Program Counter.

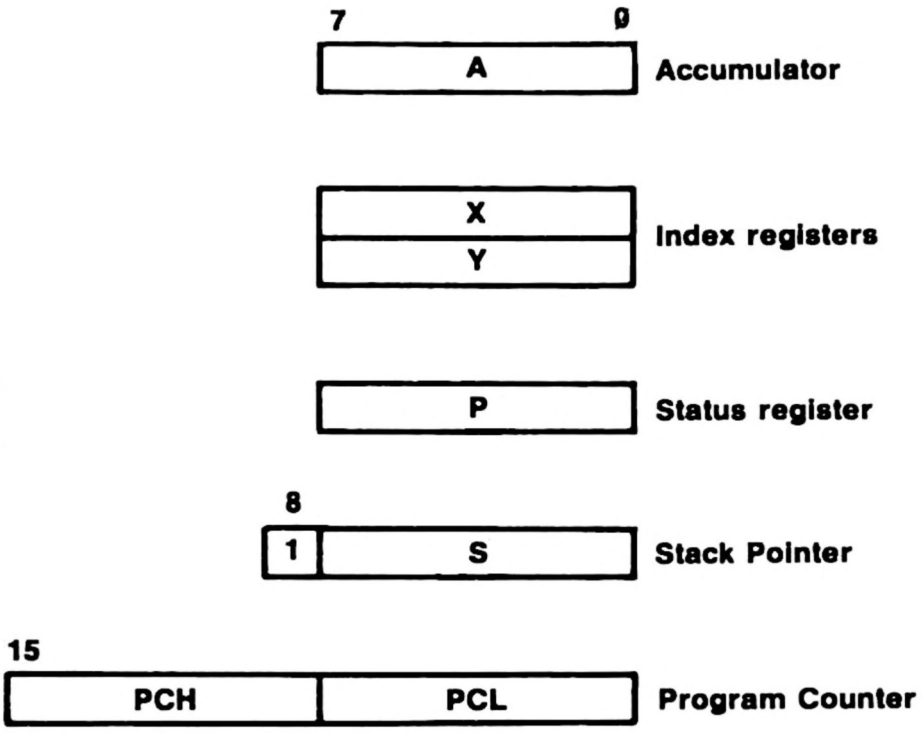


Figure 4.1 The registers—a typical programming model.

THE ACCUMULATOR

We have already mentioned the accumulator (or 'A' register) several times in the opening chapter. As you may have already gathered, the accumulator is the main register of the 6502, and like most of the other registers it is eight bits wide. This means that it can hold a single byte of information at any one time. Being the main register, it has the most instructions associated with it, and its principle feature is that all arithmetic and logical operations are carried out through it.

The accumulator's associated instructions are listed in Table 4.1. It is not absolutely vital to be familiar with these at present, but they are included now as an introduction.

Table 4.1

Accumulator instructions			
ADC	Add with carry	PHA	Push accumulator
AND	Logical AND	PLA	Pull accumulator
ASL	Arithmetic shift left	ROL	Rotate left
BIT	Compare memory bits	ROR	Rotate right
CMP	Compare to accumulator	SBC	Subtract with carry
EOR	Logical EOR	STA	Store the accumulator
LDA	Load the accumulator	TAX	Transfer accumulator to X register
LSR	Logical shift right	TAY	Transfer accumulator to Y register
ORA	Logical OR	TXA	Transfer X register to accumulator
		TYA	Transfer Y register to accumulator

THE INDEX REGISTERS

There are two further registers in the 65Ø2 which can hold single byte data. These are the *X register* and the *Y register*. They are generally termed the 'index registers', because they are very often used to provide an 'offset' or index from a specified base address. They are provided with direct increment and decrement instructions—something the accumulator lacks—so are also quite often used as counters. However, it is not possible to perform arithmetic or logical operations in either index register, but there are instructions to transfer the contents of these registers into the accumulator and vice versa.

The instructions associated with both registers are given in Table 4.2.

Table 4.2

X register instructions		Y register instructions	
CPX	Compare X register	CPY	Compare Y register
DEX	Decrement X register	DEY	Decrement Y register
INX	Increment X register	INY	Increment Y register
LDX	Load the X register	LDY	Load the Y register
STX	Store the X register	STY	Store the Y register
TAX	Transfer accumulator to X reg.	TAY	Transfer accumulator to Y reg.
TXA	Transfer X reg. to accumulator	TYA	Transfer Y reg. to accumulator
TSX	Transfer Status to X register		
TXS	Transfer X register to Status		

THE PROGRAM COUNTER

The Program Counter is the 65Ø2's address book. It nearly always contains the address in memory where the next instruction to be executed sits. Unlike the other registers, it is a 16 bit register, consisting physically of two 8 bit registers. These two are generally referred to as Program Counter High (PCH) and Program Counter Low (PCL).

5 A Poke at Machine Code

Now that we have got some of the basics out of the way, why don't we write our first machine code program—after all, that's what this book is all about!

As you are no doubt aware, REM statements can be used to give additional information to the user without affecting the program. In the following listings REM statements with *two asterisks* indicate subroutine or program titles, and REMs with *one asterisk* explain what the following machine code does. The assembly language instructions are included as REMs (without asterisks) alongside the machine code DATA statements. Enter the following—you can omit the REM statements if you like.

Program 1

```
 10  REM * * MACHINE CODE DEMO * *
 20  REM * * PLACE 'A' ON SCREEN * *
 30  CODE = #400
 40  FOR LOOP = 0 TO 5
 50     READ BYTE
 60     POKE CODE + LOOP, BYTE
 70  NEXT LOOP
 80
 90  REM * * MACHINE CODE DATA * *
 95  REM * Load accumulator with code for 'A' *
100  DATA #A9, #41          : REM LDA @ASC "A"
105  REM * Store accumulator at location #BF50 *
110  DATA #8D, #50, #BF     : REM STA #BF50
115  REM * Return from this subroutine to BASIC *
120  DATA #60               : REM RTS
130
140  REM * * EXECUTE MACHINE CODE * *
150  CLS
160  CALL (CODE)
```

The function of this short program is to place the letter 'A' on the screen. Nothing spectacular, but the program does incorporate various features that will be common to all your future machine code programs. The meaning of each line is as follows:

Line 30	Declare a variable called CODE to denote where the machine code is placed.
Line 40	Set up a data-reading loop.
Line 50	Read one *byte* of machine code data.
Line 60	POKE byte value into memory.
Line 70	Repeat loop until finished.
Line 100	Machine code data.
Line 110	Machine code data.
Line 120	Machine code data.
Line 150	Clear screen.
Line 160	Execute the machine code.

To see the effect of the program, just type in RUN, hit the RETURN key and voila—the A is there!

Let's try to answer two questions that come immediately to mind, namely:

1. Where can machine code be stored?
2. How can machine code be entered?

CODE—THE PROGRAM COUNTER

It should be fairly obvious that the machine code we write has to be stored somewhere in memory. In all the programs in this book I have used the BASIC variable 'CODE' as a pointer to the start address of the memory where the machine code is to be placed. (CODE acts, in effect, rather like the processor's own Program Counter). You may wish to use your own variable name—and this is perfectly acceptable. For example, you might consider that PC or even MACHINECODE are more appropriate names for the start of the code. It does not really matter. What matters is that you get into the habit of using the same variable name in all your programs, and thus avoid ambiguity.

The value given to CODE must be chosen with care. It would be easy enough to allocate an area which causes the machine code to overwrite another program or even the assembly program itself! In Program 1 CODE is set to #400 using the normal variable assignment statement:

CODE = #400

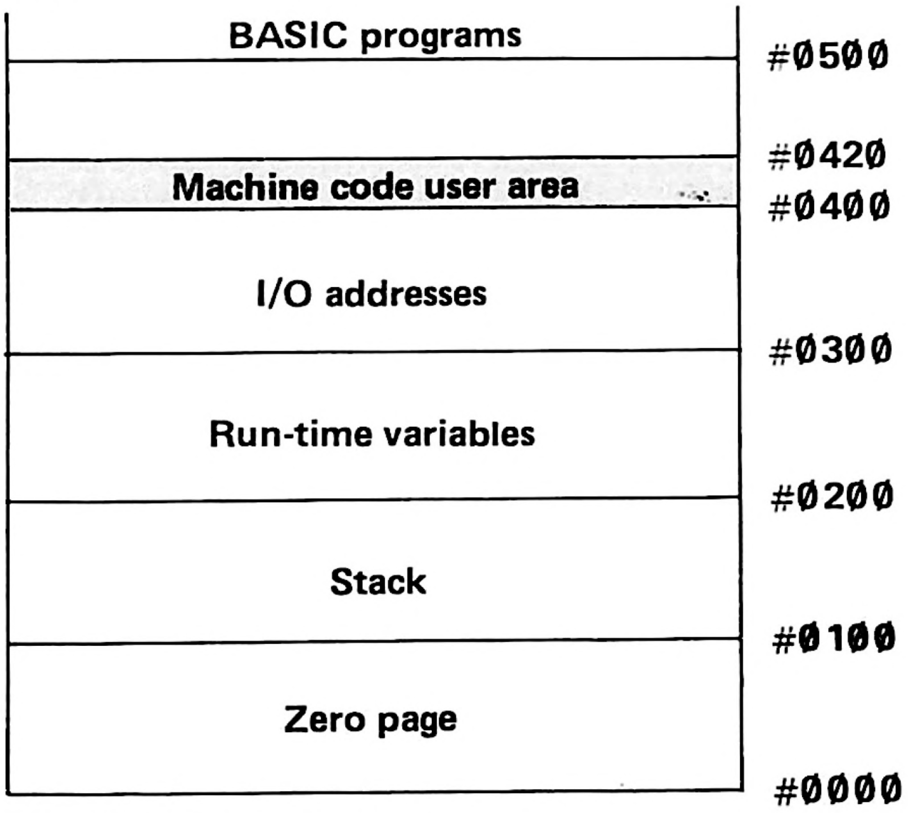


Figure 5.1 The machine code user area

and the 6 bytes of machine code are stored here—or more correctly—in the 6 bytes starting at #400 (#400 to #405). If you look at Figure 5.1 you will notice that this area has been reserved specifically for machine code programs. There are 32 bytes available here (#400 to #420) so, provided programs do not exceed this length, no problems should occur when using this area of memory.

Quite often though machine code programs do exceed 32 bytes in length so an alternative storage area is required; there are two alternatives.

Firstly, if you are only using TEXT mode then the area of memory normally reserved for the HIRES mode can be employed. A quick look at Figure 5.2 shows that the memory associated with HIRES mode extends from #9800 to #B400—a massive 7168 bytes or more simply 7K—and this should be more than ample for even the longest of machine code programs. However, using the GRAB command after placing your code in this area is perilous.

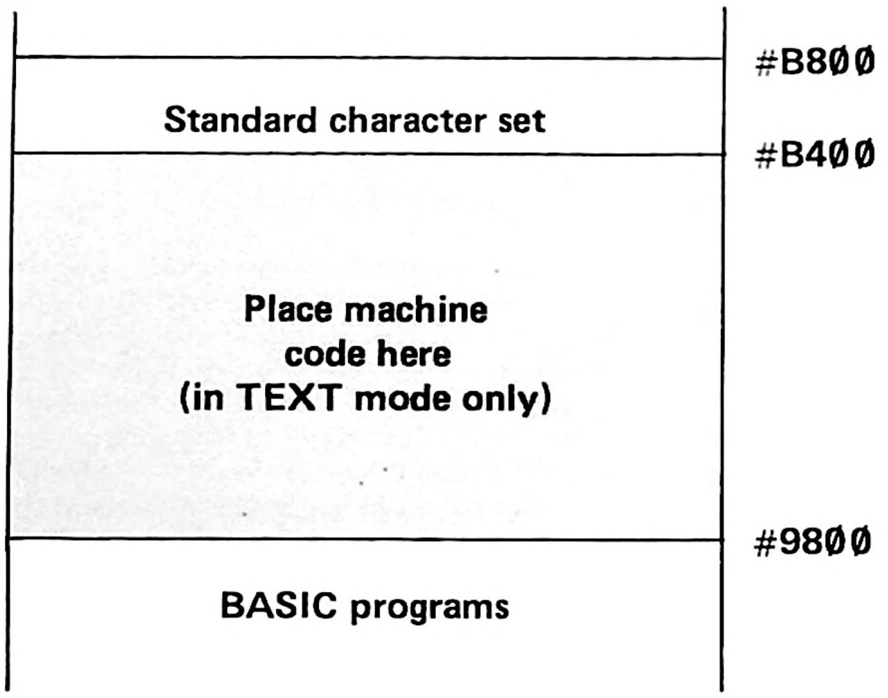


Figure 5.2 Place machine code in HIRES area

Perhaps the best and safest method is to lower the top of the user BASIC program area and place the machine code above it. This protects the machine code from being overwritten because normal BASIC programs cannot extend beyond the value assigned to the top of the user area, and it is also completely independent of the memory associated with both the HIRES and TEXT modes.

The pseudo variable associated with the top of the user area is HIMEM—short for high memory point. This variable may only be written to; it cannot be read directly. There are four bytes near the beginning of the Oric's memory map (in what's known as zero page—but more on that later), which seem to be associated with HIMEM. The two sets of bytes are:

#A2 and #A3

#A6 and #A7

but their exact relationship is not clear.

Looking at Figure 5.3, we can see that the top of the user program area is located at #9800. All we need to do is to calculate how many bytes the machine code will require, add some more for safety and reset HIMEM. For instance, suppose we need about 500 bytes or so. Converting this to hex and subtracting it from #9800, we obtain a rounded figure of #9600. To reset HIMEM to this value enter:

```
HIMEM #9600
```

Note that no '=' sign is needed—just a space. Once set, only GRAB will alter its value—executing NEW or even performing a reset will not affect it.

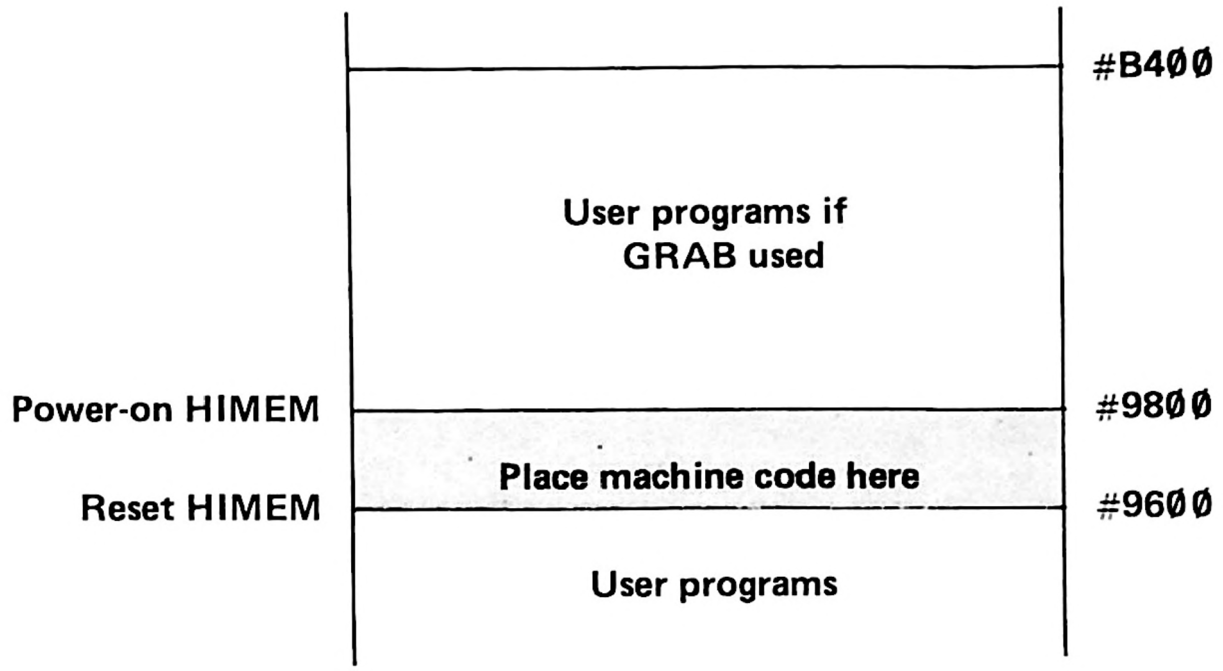


Figure 5.3 Place machine code above HIMEM

Using the space above HIMEM has the added advantage that it frees the machine code area. This can then be used by your machine code programs as a data and address storage area.

To summarize then, the following areas can be considered reasonably 'safe' for machine code:

1. User machine code area (#400 to #420).
2. Memory normally reserved for HIRES mode (#9800 to #B400).
3. Above the reset value of HIMEM.

ENTERING MACHINE CODE

Program 1 (page 18) demonstrates the most obvious method of entering a machine code program. The hex is placed in a series of data statements which are subsequently READ from within a FOR . . . NEXT loop and POKEd into memory using the loop counter (LOOP) as an offset from the machine code base address (given by CODE). This ensures that the bytes are placed in consecutive memory locations.

Notice also in Program 1, and for that matter in all the programs in this book, each machine code instruction is placed in a separate DATA statement and followed by a REM statement which depicts mnemonically the operation of that particular opcode. For example:

```
100  DATA #A9, #41: REM LDA @ASC"A"
```

It is a good idea to get into the habit of doing this simply because it adds to the program's readability. This is particularly important if you come back to the program several weeks after writing it, when you will almost certainly have forgotten what you originally wrote. Being faced with a DATA line containing nothing but hex numbers will do nothing to jog your memory:

```
123  DATA #A9, #70, #85, #22, #8D, #50, #90, #60
```

if you see what I mean!

One final point regarding the loop count. When entering this, count the total number of hex bytes and subtract one. (Remember that the loop counter itself should start from '0' to ensure that the very first byte is placed at the address specified by CODE—CODE + LOOP = #400 + 0 = #400.)

Although the above method of entering machine code is probably the best, it is somewhat long-winded. An easier way involves the use of a machine code monitor program. This allows machine code to be entered into memory by typing in hex numbers in response to certain prompts. A simple, though adequate example of such a monitor is given in Program 2.

Program 2

```
 10  REM * * ORIC MACHINE CODE MONITOR * *
 20  CLS
 30  INK 3: PAPER 4
 40  PRINT "ORIC MONITOR": PRINT
 50  INPUT "Enter Assembly Address :"; A$
 60  ADDR = VAL(A$)
 70  REPEAT
 80    PRINT HEX$(ADDR); " : #";
 90    GET B$
100    IF B$ = "S" GOTO 250: REM * STOP *
110    IF ASC(B$) > ASC("F") GOTO 90
120    B$ = "#" + B$: HIGH = VAL(B$)
130    IF HIGH > 15 GOTO 90
140    IF HIGH < 0 GOTO 90
150    PRINT RIGHT$(B$, 1);
160    GET C$
170    IF ASC(C$) > ASC("F") GOTO 160
180    C$ = "#" + C$: LOW = VAL(C$)
190    IF LOW > 15 GOTO 160
200    IF LOW < 0 GOTO 160
210    PRINT RIGHT$(C$, 1)
220    BYTE = HIGH * 16 + LOW
230    POKE ADDR, BYTE
240    ADDR = ADDR + 1
250  UNTIL B$ = "S"
```

Enter and RUN the program. After it displays the heading you are asked to enter the 'assembly address'. This is the address from which you wish the machine code to be stored, and is the value you would normally have assigned to CODE. The address can be entered as a decimal number, or as a hex value if preceded by a '#'. On hitting RETURN the first address is displayed in hex notation and is followed by a further '#'. All you now have to do is to type in the hex digits (minus the hash which is already there!). After you type the second digit of the byte, it is POKEd into the appropriate memory location and the next address is displayed.

To leave the monitor program just type 'S' (for Stop!) when the next address is displayed. The program checks for (and rejects) illegal hex values. A typical RUN from the monitor program is shown in Figure 5.4.

```
ORIC MONITOR

Enter Assembly Address :?#400
#400  :  #A9
#401  :  #41
#402  :  #8D
#403  :  #50
#404  :  #BF
#405  :  #60
#406  :  #S

Ready
```

Figure 5.4   A typical monitor run.

CALLING MACHINE CODE

To execute a machine code program the BASIC statement CALL is used. To tell the BASIC interpreter just where the machine code is located, the CALL statement must be followed by a label or address. So, to execute the machine code generated by the assembly program type in either:

```
CALL(CODE)
```

which is the label name which marks the start of the assembly program, or:

```
CALL(#400)
```

which is the start address of the machine code itself.

GETTING IT TAPED

It is a very good idea, as a matter of routine, to get into the habit of saving your machine code programs on tape BEFORE you actually RUN them. This may seem a bit back to front because you normally would not do this in BASIC until you had RUN, tested and debugged the program. The trouble with running a machine code program for the first time, though, is that if it does contain any bugs, it will almost certainly cause the Oric to 'hang-up' and the only way out of this is to switch it off (and then back on again) at the mains, and start again. RESET has no effect whatsoever. If your machine code does fail in this way, and you've saved it on tape, all you have to do is to CLOAD it back in again and swat the bug out!

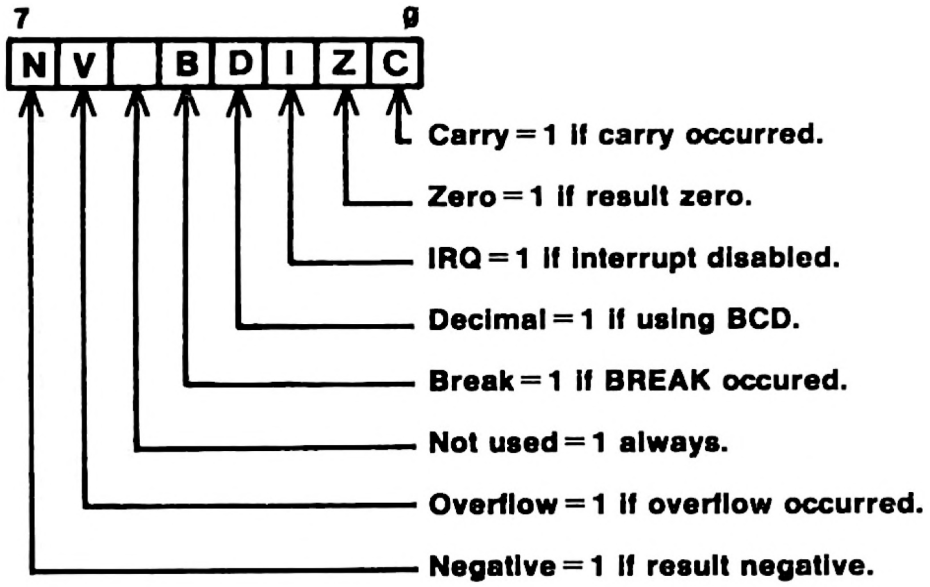
Full details of loading and saving programs and blocks of memory on to tape are described in Chapter 11 of the *Manual.*

6 Status Symbols

THE STATUS REGISTER

The Status register is unlike the various 'other' registers of the 65Ø2. When using it, we are not really concerned with the actual hex value it contains, but more with condition or state of its individual bits. These individual bits are used to denote or *flag* certain conditions as and when they occur during the course of a program. Of the register's eight bits, only seven are in use—the remaining bit (bit 5) is permanently set. (In other words it always contains a 1.)

Figure 6.1 shows the position of the various flags, each of which is now described in detail.



7
Ø
N
V
B
D
I
Z
C
Carry = 1 if carry occurred.
Zero = 1 if result zero.
IRQ = 1 if interrupt disabled.
Decimal = 1 if using BCD.
Break = 1 if BREAK occured.
Not used = 1 always.
Overflow = 1 if overflow occurred.
Negative = 1 if result negative.

Figure 6.1 Status register flags.

Bit 7: The Negative flag (N)

In signed binary, the Negative flag is used to determine the sign of a number. If the flag is set (N = 1) the result is negative. If the flag is clear (N = Ø) the result is positive.

However a whole host of other instructions condition this particular flag, including all the arithmetic and logical instructions. In general, the most significant bit of the result of an operation is copied directly into the N flag.

Consider the following two operations:

```
LDA @#8Ø        \ load accumulator with #8Ø
```

This will set the Negative flag (N = 1) because #8Ø = 1ØØØØØØØ in binary. Alternatively:

```
        LDA @#7F        \ load accumulator with #7F
```

will clear the Negative flag (N = 0) because #7F = 01111111 in binary. There are two instructions which act on the state of the N flag—these are:

BMI	Branch on minus (N = 1)
BPL	Branch on plus (N = 0)

More on these later.

Bit 6: The Overflow flag (V)

This flag is probably the least used of all the Status register flags. It is used to indicate if a carry occurred from bit 6 during an addition, or if a borrow occurred to bit 6 in a subtraction. If either of these events took place the flag is set (V = 1).

Look at the following two examples:
First, #09 + #07:

```
    (#09)          0 0 0 0 1 0 0 1
    (#07)        + 0 0 0 0 0 1 1 1
                   ---------------
    (#10)          0 0 0 1 0 0 0 0
                   ^
                   └——No overflow from bit 6 therefore V = 0.
```

Second, #7F + #01:

```
    (#7F)          0 1 1 1 1 1 1 1
    (#01)        + 0 0 0 0 0 0 0 1
                   ---------------
    (#80)          1 0 0 0 0 0 0 0
                   ^
                   └——Overflow has occurred from bit 6 therefore V = 1.
```

If we were using signed binary this addition would give a result of −128, which is of course incorrect. However this fact is flagged and so the result can be corrected as required.

Bit 5

This bit is not used and is permanently set.

Bit 4: The Break flag (B)

This flag is set whenever a BREAK occurs, otherwise it will remain clear. This may seem a bit odd at first, because surely we will know when a BREAK occurs. However, it is possible to generate a BREAK externally by something called an *Interrupt*, and this flag is used to help distinguish between these 'BREAKs'.

Bit 3: The Decimal flag (D)

This flag tells the processor just what type of arithmetic is being used. If it is cleared (by CLD), as is usual, then normal hexadecimal operation occurs. If set (by SED) all values will be interpreted as Binary Coded Decimal.

Bit 2: The Interrupt flag (I)

We mentioned interrupts above in the description of the Break flag. Suffice to say now, that the flag is set (I = 1) when the IRQ interrupt is disabled, and is clear (I =0) when IRQ interrupts are permitted.

Bit 1: The Zero flag (Z)

As its name implies, the flag is used to show whether or not the result of an operation is zero. If the result is zero the flag is set (Z = 1), otherwise it is cleared (Z = 0). It is true to say that the Zero flag is conditioned by the same instructions as the Negative flag. Executing:

```
LDA @0          \ load accumulator with zero
```

will set the Zero flag (Z = 1) but:

```
LDX @#FA        \ load X register with #FA
```

will clear the Zero flag (Z = 0).

Bit 0: The Carry flag (C)

We have already seen that adding two bytes together can result in carries occurring from one bit to another. What happens if the carry is generated by the most significant bits of an addition?

For example, when adding #FF + #80:

```
(#FF)           1 1 1 1 1 1 1 1
(#80)         + 1 0 0 0 0 0 0 0
               ----------------
(#7F)        (1)0 1 1 1 1 1 1 1
              └─Carry over from bits 7
```

the result is just too large for eight bits, an extra ninth bit is required. The Carry flag acts as this ninth bit.

If the Carry flag is clear at the start of an addition (C = 0) and set on completion (C = 1) the result is greater than 255. It follows that if the flag is set (C = 1) before a subtraction and clear on completion (C = 0), the value being subtracted was larger than the original value. Two instructions are available for direct use on the Carry flag:

CLC	Clear Carry flag (C = 0)
SEC	Set Carry flag (C = 1)

Two instructions are also provided to act on the condition of the Carry flag.

BCC	Branch on Carry clear (C = 0)
BCS	Branch on Carry set (C = 1)

7 Addressing Modes I

The 6502 has quite a small instruction set when compared with some of its fellow microprocessors—in fact it has a basic clique of just 56 instructions. However, many of these can be used in a variety of ways, which effectively increases the range of operations to 152. The way in which these instructions are interpreted is determined by the *addressing mode* used, some examples of which are given here:

Addressing mode	Mnemonic example	Opcode	Operand(s)
Immediate	LDA @#FF	A9	FF
Zero page	LDA #70	A5	70
Zero page indexed	LDA #70, X	B5	70
Absolute	LDA #1500	AD	00 15
Indirect pre-indexed	LDA (#70, X)	A1	70
Indirect post-indexed	LDA (#70), X	B1	70
Absolute indexed	LDA #1500, X	BD	00 15

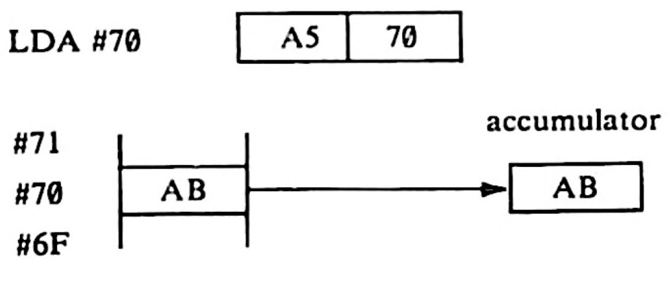
All seven of these instructions load the accumulator—but in each case the data loaded is obtained from a different source as defined by the opcode. This, as you may have noticed, is different in each case.

For the time being we shall only look at the first two of these addressing modes, immediate and zero page, both of which we have used several times already.

ZERO PAGE ADDRESSING

Zero page addressing is used to specify addresses in the first 256 bytes of RAM where data which has to be loaded into a specified register could be located. Because the high byte of the address will always be #00 it is omitted, and therefore the instruction and address require just two bytes of memory.

Operation: DATA #A5, #70



In the above example, LDA #7Ø, the contents of location #7Ø (in this case #AB) are loaded into the accumulator.

The use of zero page needs some care as this area is used as a scratchpad by the BASIC interpreter for storing addresses and performing calculations. No problems should occur if your program is written exclusively in machine code. However, particular care is needed if your program is written in a hybrid of BASIC and machine code, because it is quite possible that your machine code could corrupt zero page data being held by the interpreter (FOR . . . NEXT loop counters for instance). Unfortunately Oric have not made any documentation available on this sensitive area of memory, so if possible, use the user area between #4ØØ and #42Ø as a data storage area. If you must use zero page, then test your machine code fully to ensure it does not corrupt the calling BASIC. (If it does, then reCLOAD it, and try using alternative locations.)
The instructions associated with zero page addressing are shown in Table 7.1.

Table 7.1

Zero page addressing instructions

ADC	Add with carry	LDX	Load X register
AND	Logical AND	LDY	Load Y register
ASL	Arithmetic shift left	LSR	Logical shift right
BIT	Bit test	ORA	Logical OR
CMP	Compare accumulator	ROL	Rotate left
CPX	Compare X register	ROR	Rotate right
CPY	Compare Y register	SBC	Subtract with carry
DEC	Decrement memory	STA	Store accumulator
EOR	Logical EOR	STX	Store X register
INC	Increment memory	STY	Store Y register
LDA	Load accumulator		

IMMEDIATE ADDRESSING

This form of addressing is used to load the accumulator, or the index registers, with a specific value which is known at the time of writing the program. The 65Ø2 will know from the opcode that the byte following is in actual fact data and not an address. However, to remind *us* of the fact, and to assist us when we are writing the initial assembler, we can precede the data byte with a '@' sign (this shares the '2' key on the Oric keyboard). Only single byte values can be specified because the register size is limited to just eight bits.
If we wish our machine code program to load the accumulator with 255, we can include the following two-byte sequence in our program:

```
DATA #A9, #FF: REM LDA @#FF
```

where #A9 is the 'load the accumulator immediate' code.
Similarly, the X and Y registers can be loaded immediately with:

```
DATA #A2, #41 : REM LDX @"A"

DATA #AØ, #FA : REM LDY @#FA
```

where A2 and AØ are the immediate codes for loading the X and Y registers.

Operation:

```
LDA #255       | A9 | FF |

Accumulator    | FF |
```

Program 3 uses both zero page and immediate addressing to place an exclamation mark on the screen.

Program 3

```
 10  REM * * ZERO PAGE ADDR. * *
 20  CODE = #400
 30  FOR LOOP = 0 TO 9
 40  READ BYTE
 50  POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 81  REM * Load X with ASCII code for X *
 82  REM * Store code in location #22 *
 83  REM * Load accumulator with contents of #22 *
 84  REM * Store accumulator in screen memory *
 85  REM * Return back to BASIC *
 90  DATA #A2, #21          : REM LDX @"!"
100  DATA #86, #22          : REM STX #22
110  DATA #A5, #22          : REM LDA #22
120  DATA #8D, #AA, #BB : REM STA SCREEN
130  DATA #60               : REM RTS
140
150  REM * * SET UP & EXECUTE * *
160  CLS
170  CALL (CODE)
```

8 Bits and Bytes

LOAD, STORE AND TRANSFER

To enable memory and register contents to be altered and manipulated, three sets of instructions are provided.

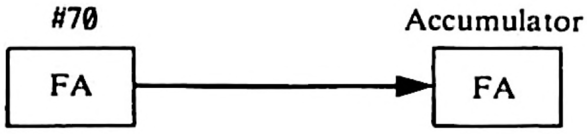
Load instructions

The process of placing memory contents into a register is known as *loading*, some examples of which we have already seen. To recap however, these are the three load instructions:

LDA	Load accumulator
LDX	Load X register
LDY	Load Y register

All of these instructions may be used with immediate addressing, but when dealing with memory locations, it is more correct to say that the contents of the specified address are *copied* into the particular register, as the source location is not altered in any way.

For example, with LDA #70, the contents of location #70 (in this case FA) are copied into the accumulator, location #70 is not altered:



The Negative and Zero flags of the Status register are conditioned by the load operation.

Store instructions

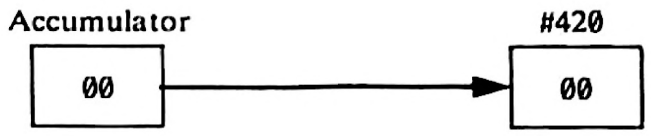
The reverse process of placing a register's contents into a memory location, is known as *storing*. There are three store instructions:

STA	Store accumulator
STX	Store X register
STY	Store Y register

The register value is unaltered and no flags are conditioned.

Example:

```
LDA @#0
STA #420
```



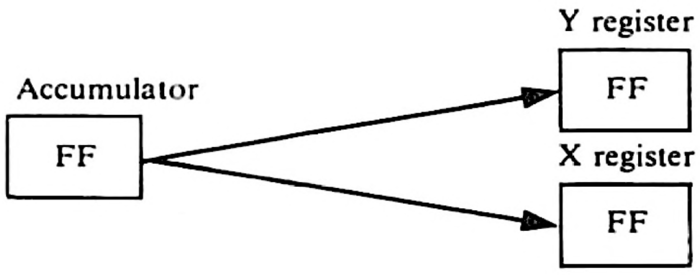
Transfer instructions

Instructions are provided to allow the contents of one register to be copied into another—this is known as *transferring*. The Negative and Zero flags are conditioned according to the data being transferred. There are four instructions controlling transfers between the index registers and the accumulator.

TXA	Transfer X register to accumulator
TAX	Transfer accumulator to X register
TYA	Transfer Y register to accumulator
TAY	Transfer accumulator to Y register

Example:

```
LDA @#FF
TAY
TAX
```



Unfortunately, you cannot transfer directly between the X and Y registers, you have to use the accumulator as an intermediate store.

```
YTOX  TYA                           \ Y into accumulator
      TAX                           \ accumulator into X
```

Similarly:

```
XTOY  TXA                           \ X into accumulator
      TAY                           \ accumulator into Y
```

This form of single byte operation is known as *implied addressing* because the information is contained within the instruction itself.

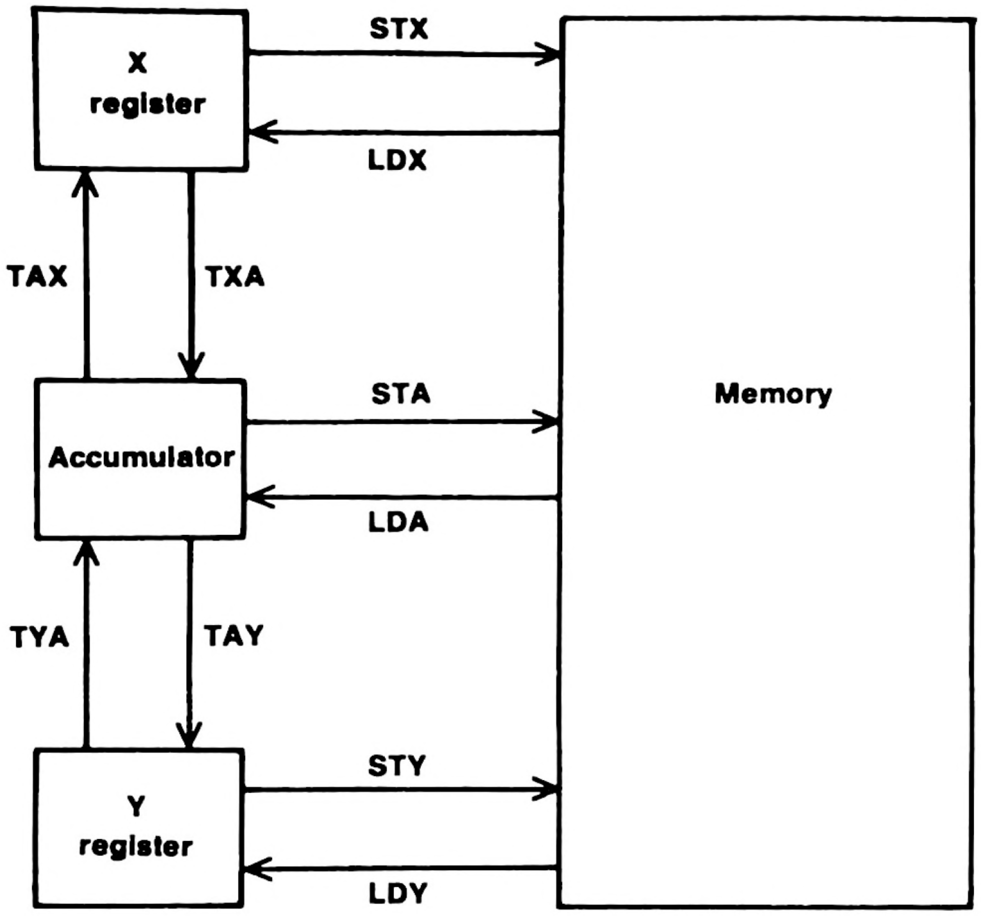


Figure 8.1 Load, store and transfer instruction flow.

PAGING MEMORY

We have seen that the Program Counter consists of two eight bit registers, giving a total of 16 bits. If all these bits are set, 11111111 11111111, the value obtained is 65536, or #FFFF. Therefore the maximum addressing range of the 6502 is #0000 through to #FFFF. This range of addresses is implemented as a series of *pages* and the page number is given by the contents of PCH. It follows that PCL holds the address of the location on that particular page.

As Figure 8.2 illustrates, each page of memory can be likened to a page of a book. This book has 256 pages labelled in hex format from #00 to #FF. Each individual page is ruled into 256 lines which in turn are labelled (from top to bottom) #00 to #FF.

Thus the address #FFFF refers to line #FF on page #FF, the very last location in the Oric's memory map! Unlike conventional books, memory begins with page #00 which is more affectionately known as *zero page*. Owing to the 6502's design zero page is very important, as we shall see when we take a further look at addressing modes in Chapter 10.

Although we have referred to the Oric's memory map as a series of pages, it is more frequently talked of in terms of 'K'. For example, Orics are supplied with either 16K or 48K of memory. The term 'K' is short for kilo, but unlike its metric counterpart, one kilo of memory, or a kilobyte, consists of 1024 bytes and not 1000 bytes. This slightly higher value is chosen because it is divisible by 256 and corresponds to exactly four pages of memory (4 × 256 = 1024). The total memory map therefore encompasses 64K as the following kilobyte calculator (written in BASIC!) will confirm.

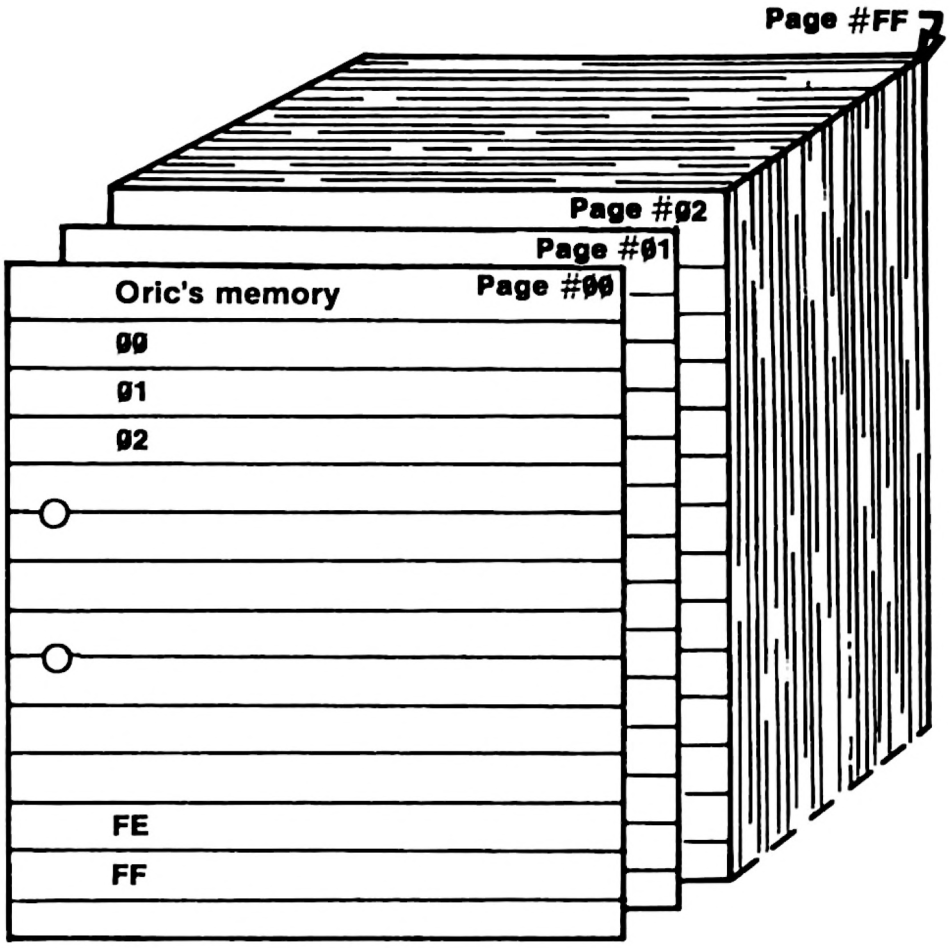


Figure 8.2 Pages of the Oric's memory.

Program 4

```
10 REM * * KCALC * *
20 CLS
30 REPEAT
40 INPUT "K value:"; K
50 BYTES = 1024 * K
60 PRINT "K value is"; BYTES; "Decimal";
70 PRINT HEX$(BYTES); "Hex"
80 UNTIL 0
```

9 Arithmetic in Machine Code

We can now put some of the basic principles we have encountered in the opening chapters to some more serious use—the addition and subtraction of numbers. These two procedures are fundamental to machine code and will generally find their way into most programs.

ADDITION

Two instructions facilitate addition, they are:

CLC	Clear Carry flag
ADC	Add with carry

The first of these instructions, CLC, simply clears the Carry flag (C = 0). This will generally be performed at the very onset of addition, because the actual addition instruction, ADC, produces the sum of the accumulator, the memory byte referenced and the Carry flag. The reason for doing this will become clearer after we have looked at some simple addition programs. Enter Program 5.

Program 5

```
 10 REM * * SIMPLE ADD * *
 20 CODE = #400
 30 FOR LOOP = 0 TO 7
 40    READ BYTE
 50    POKE CODE + LOOP, BYTE
 60 NEXT LOOP
 70
 80 REM * * M/C DATA * *
 90
100 DATA #18                 : REM CLC
110 DATA #A9, #07            : REM LDA @#7
120 DATA #69, #03            : REM ADC @#3
130 DATA #85, #70            : REM STA #70
140 DATA #60                 : REM RTS
```

```
150
160 CALL (CODE)
170 PRINT "RESULT IS :";
180 PRINT PEEK (#70)
```

As can be seen in line 110, this program loads 7 into the accumulator using immediate addressing. Immediate addressing is used again in line 120 to add 3 to the accumulator value. The result, which is in the accumulator, is then stored at #70. Line 160 executes the assembled machine code, and the result (if you're quick with your fingers you'll know it's 10!), is printed out. RUN the program to see its effect, and try substituting your own values in lines 110 and 120.

Re-type line 100 thus:

```
100 DATA #38                    : REM SEC
```

This instruction will set the Carry flag when the program is next executed. Reset lines 110 and 120 (if you have altered them), and RUN the program again. The result is now 11. The reason being that the Carry flag's value is taken into consideration during ADC (add with carry) and this time its value is 1.

	Accumulator	+	memory	+	carry	=	result
CLC	7	+	3	+	0	=	10
SEC	7	+	3	+	1	=	11

Again you might like to try your own immediate values—you'll find the result is always one greater than expected.

This program is quite wasteful both in terms of memory used and time taken for execution. If we know the values to be added together beforehand, then it is more efficient to add them together first. The machine code part of the program can then be incorporated into just two lines:

```
DATA #A9, #0A                   : REM LDA @10
DATA #85, #22                   : REM STA #22
```

Program 6 is a general purpose single byte addition program.

Program 6

```
10  REM * * GENERAL PURPOSE SINGLE BYTE ADD * *
20  CODE = #400
30  FOR LOOP = 0 TO 7
40      READ BYTE
50      POKE CODE + LOOP, BYTE
60  NEXT LOOP
70
80  REM * * M/C DATA * *
90
100 DATA #18                    : REM CLC
110 DATA #A5, #70               : REM LDA #70
120 DATA #65, #71               : REM ADC #71
```

```
130 DATA #85, #72                    : REM STA #72
140 DATA #60                         : REM RTS
150
160 REM * * SET UP & EXECUTE * *
170 CLS
180 INPUT "FIRST NUMBER";NUM1
190 POKE #70, NUM1
200 INPUT "SECOND NUMBER";NUM2
210 POKE #71, NUM2
220 CALL (CODE)
230 PRINT "RESULT IS :";
240 PRINT PEEK (#73)
```

RUN the program a few times entering low numerical values in response to the program's prompts.

Now enter 128 and 128, as your inputs. The result is 0, why? The reason is that the answer, 256, is too big to be held in a single byte:

```
   128         #80         1 0 0 0 0 0 0 0
 + 128       + #80       + 1 0 0 0 0 0 0 0
   ---         ---         ---------------
   256        #100      (1)0 0 0 0 0 0 0 0
```

and as can be seen, a carry has been produced by the bit overflow from adding the two most significant bits. As the Carry flag was initially cleared before the addition, it will now be set, signalling the fact that the result is too large for a single byte.

This principle is used when summing multibyte numbers, and is illustrated by Program 7, which adds two double byte numbers.

Program 7

```
 10 REM * * DOUBLE BYTE ADD * *
 20 CODE = #400
 30    FOR LOOP = 0 TO 13
 40    READ BYTE
 50    POKE CODE + LOOP, BYTE
 60 NEXT LOOP
 70
 80 REM * * M/C DATA * *
 90
100 DATA #18                         : REM CLC
110 DATA #A5, #70                    : REM LDA #70
114 DATA #65, #72                    : REM ADC #72
115 DATA #85, #74                    : REM STA #74
121 DATA #A5, #71                    : REM LDA #71
122 DATA #65, #73                    : REM ADC #73
130 DATA #85, #75                    : REM STA #75
```

```
140 DATA #60                        : REM RTS
150
160 REM * * SET UP & EXECUTE * *
170 CLS
180 INPUT "FIRST NUMBER";NUM1
190 DOKE #70, NUM1
200 INPUT "SECOND NUMBER";NUM2
210 DOKE #72, NUM2
220 CALL (CODE)
230 PRINT "RESULT IS :";
240 PRINT DEEK (#74)
```

This routine will produce correct results for any two numbers whose sum is less than 65536 (#FFFF) which is the highest numerical value that can be held in two bytes of memory.

Note that the Carry flag is cleared at the onset of the machine code itself. If any carry should occur when adding the two low bytes together, it will be transferred over to the addition of the two high bytes.

SUBTRACTION

The two associated instructions are:

SEC	Set Carry flag
SBC	Subtract, borrowing carry

The operation of subtracting one number from another (or finding their difference) is the reverse of that used in the preceding addition examples. Firstly the Carry flag is set (C = 1) with SEC, and then the specified value is subtracted from the accumulator using SBC. The result of the subtraction is returned in the accumulator.

The following program performs a single byte subtraction:

Program 8

```
10 REM * * SINGLE BYTE SUBTRACTION * *
20 CODE = #400
30 FOR LOOP = 0 TO 7
40    READ BYTE
50    POKE CODE + LOOP, BYTE
60 NEXT LOOP
70
80 REM * * M/C DATA * *
90
100 DATA #38                    : REM SEC
110 DATA #A5, #70               : REM LDA #70
120 DATA #E5, #71               : REM SBC #71
130 DATA #85, #72               : REM STA #72
```

```
140 DATA #60                    : REM RTS
150
160 REM * * SET UP & EXECUTE * *
170 CLS
180 INPUT "FIRST NUMBER ";NUM1
190 POKE #70, NUM1
200 INPUT "SECOND NUMBER ";NUM2
210 POKE #71, NUM2
220 CALL (CODE)
230 PRINT "RESULT IS :";
240 PRINT PEEK (#73)
```

RUN the program and input your own values to see the results.

You may well be wondering why the Carry flag is set before a subtraction rather than cleared. Referring back to Chapter 3, you will recall that the subtraction there was performed by adding the two's complement value. This is found by first inverting all the bits to obtain the one's complement, and then adding 1. The 6502 obtained the 1 to be added to the one's complement form, from the Carry flag. Thus we can say:

1. If the Carry flag is set after SBC, the result is positive or zero.
2. If the Carry flag is clear after SBC, the result is negative and a borrow has occurred.

Try changing line 100 to CLC and re-RUN the program. Now your results are one less than expected—the reason being that the two's complement was never obtained by the 6502, because only a '0' was available in the Carry flag to be added to the one's complement value.

To subtract double byte numbers, the Carry flag is set at the entry to the routine, and the relative bytes are subtracted and stored. The resulting program will look something like this:

Program 9

```
10 REM * * DOUBLE BYTE SUBTRACTION * *
20 CODE = #400
30 FOR LOOP = 0 TO 13
40    READ BYTE
50    POKE CODE + LOOP, BYTE
60 NEXT LOOP
70
80 REM * * M/C DATA * *
90
100 DATA #38                    : REM SEC
110 DATA #A5, #70               : REM LDA #70
120 DATA #E5, #72               : REM SBC #72
130 DATA #85, #74               : REM STA #74
140 DATA #A5, #71               : REM LDA #71
150 DATA #E5, #73               : REM SBC #73
160 DATA #85, #75               : REM STA #75
```

```
170 DATA #60                          :  REM RTS
180
190 REM * * SET UP & EXECUTE * *
200 CLS
210 INPUT "FIRST NUMBER ";NUM1
220 DOKE #70, NUM1
230 INPUT "SECOND NUMBER";NUM2
240 DOKE #72, NUM2
250 CALL (CODE)
260 PRINT "RESULT IS :";
270 PRINT DEEK (#74)
```

NEGATION

The SBC instruction can be used to convert a number into its two's complement form. This is done by subtracting the number to be converted, from zero. The following program asks for a decimal value (less than 255) and prints its two's complement value in hex:

Program 10

```
 10   REM * * 2's COMPLEMENT CONVERTER * *
 20   CODE = #400
 30   FOR LOOP = 0 TO 7
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60   NEXT LOOP
 70
 80   REM * * M/C DATA * *
 90
100   DATA #38                        :  REM SEC
100   DATA #A9, #00                   :  REM LDA @0
120   DATA #E5, #70                   :  REM SBC #70
130   DATA #85, #71                   :  REM STA #71
140   DATA #60                        :  REM RTS
150
160   REM * * SET UP & EXECUTE * *
170
180   INPUT "Number to Convert :";NUM
190   POKE #70, NUM
200   CALL (CODE)
210   PRINT "2's Complement value is :";
220   PRINT PEEK (#71)
```

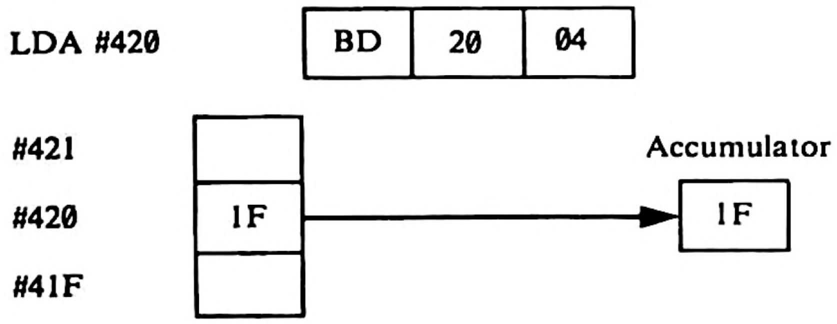
10 Addressing Modes II

Let us now take a second look at addressing modes. In the previous chapters we have seen how data can be obtained directly by an instruction using *immediate* addressing, or indirectly from a location in zero page using *zero page* addressing. We shall now see how two byte address locations can be accessed both directly and indirectly (through the all important zero page), and how whole blocks of memory can be manipulated using *indexed* addressing.

ABSOLUTE ADDRESSING

Absolute addressing works in exactly the same manner as zero page addressing, but it covers all memory locations outside zero page. The opcode is followed by two bytes which specify the address of the memory location (which can be anywhere in the range #100 to #FFFF).

Operation



As can be seen above, the operation code is followed by the address which, as always, is stored in reverse order low byte first. The contents of location #420 are copied into the accumulator when the instruction is executed. Absolute addressing was used in line 120 of Program 3 (page 29) to store the accumulator's contents into screen memory, effectively displaying the exclamation mark.

The complete list of instructions associated with absolute addressing is shown in Table 10.1.

Table 10.1

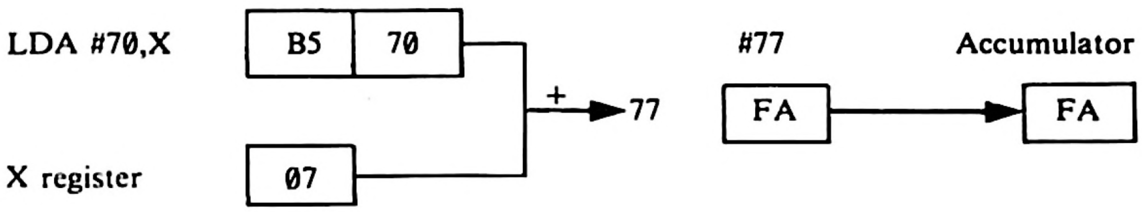
Absolute addressing instructions

ADC	Add with carry	LDA	Load accumulator
AND	Logical AND	LDX	Load X register
ASL	Arithmetic shift left	LDY	Load Y register
BIT	Bit test	LSR	Logical shift right
CMP	Compare accumulator	ORA	Logical OR
CPX	Compare X register	ROL	Rotate left
CPY	Compare Y register	ROR	Rotate right
DEC	Decrement memory	SBC	Subtract with carry
EOR	Logical EOR	STA	Store accumulator
INC	Increment memory	STX	Store X register
JMP	Jump	STY	Store Y register
JSR	Jump, save return		

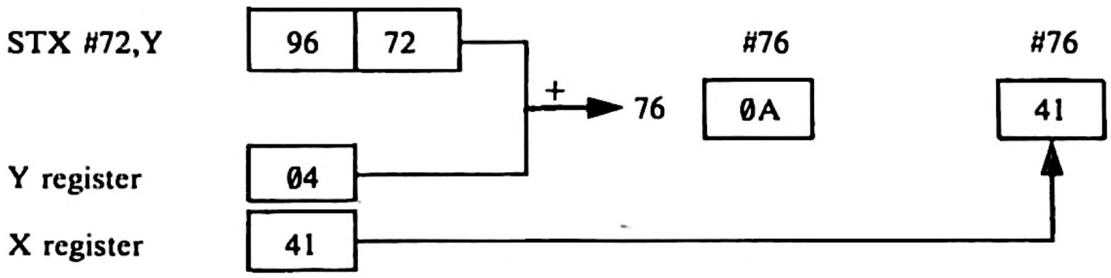
ZERO PAGE INDEXED ADDRESSING

In zero page indexed addresing, the actual address of the operand is calculated by adding the contents of either the X or Y register to the zero page address stated.

Operation:



The X register in this instance contains #07. This is added to the specified address, #70, to give the actual address, #77. The contents of location #77 (in this case FA) are then loaded into the accumulator. Similarly:



Here the Y register is used as an index to allow the contents of the X register to be stored in memory location #76. This address was obtained by adding the Y register's value, #04, to the specified value, #72. The original contents of location #76 (0A) are overwritten. The instructions associated with zero page indexing are listed in Table 10.2.

Table 10.2

Zero page indexed addressing instructions

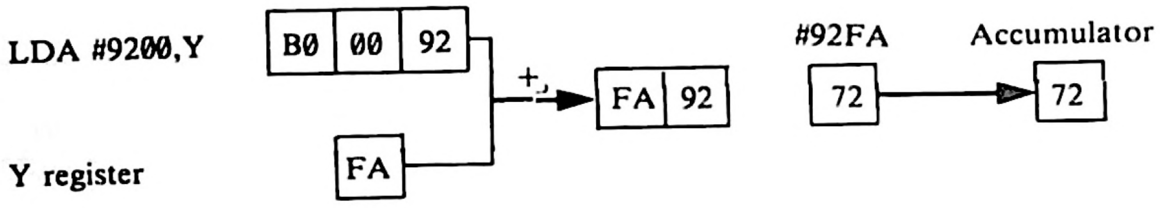
ADC	Add with carry	LDY	Load Y register
AND	Logical AND	LSR	Logical shift right
ASL	Arithmetic shift left	ORA	Logical OR
CMP	Compare	ROL	Rotate left
DEC	Decrement memory	ROR	Rotate right
EOR	Logical EOR	SBC	Subtract with carry
INC	Increment memory	STA	Store accumulator
LDA	Load accumulator	*STX	Store X register
*LDX	Load X register	STY	Store Y register

The * indicates the only commands which can use the Y register as an index. All other commands are for X register only.

ABSOLUTE INDEXED ADDRESSING

Absolute indexed addressing is like zero page indexed addressing except that the locations accessed are outside zero page. The X and Y registers may be used as required to operate with the accumulator, or each other.

Operation:



The Y register's contents (#FA) are added to the two byte address (#92ØØ) to give effective address (#92FA).

The following program demonstrates how absolute indexed addressing can be used to move a section of memory from one location to another.

Program 11

```
10  REM * * ABSOLUTE INDEXED ADDRESSING * *
20  REM * * RESET 'HIMEM #96ØØ' * *
30  CODE = #96ØØ
40  FOR LOOP = Ø TO 16
50      READ BYTE
60      POKE CODE + LOOP, BYTE
70   NEXT LOOP
80
90   REM * * M/C DATA * *
100
```

```
110  DATA #A2, #00            : REM LDX @0
120  DATA #A0, #20            : REM LDY @#20
130  DATA #BD, #AA, #BB       : REM LDA #BBAA, X
140  DATA #09, #20            : REM ORA @#20
150  DATA #9D, #EA, #BC       : REM STA #BCEA, X
160  DATA #E8                 : REM INX
170  DATA #88                 : REM DEY
180  DATA #D0, #F4            : REM BNE -12
190  DATA #60                 : REM RTS
200
210  REM * * SET UP & EXECUTE * *
220  CLS
230  PRINT "ABSOLUTE INDEXED ADDRESSING"
240  GET A$ : REM Press a key
250  CALL (CODE)
```

When RUN, the message of line 230 is printed on to the TEXT mode display. The program then waits for a key to be pressed before calling CODE. The X register acts as the offset counter and the Y register as the loop counter—lines 110 and 120 initialize them. The byte at 'the top of the screen + X' is then loaded into the accumulator using absolute indexed addressing (line 130). It is then converted into a lower case ASCII character by logically ORing its present ASCII value with #20. This forces bit 5 to one (which is the binary difference between upper case and lower case ASCII characters). For example:

```
ASC "A" = #41   0 1 0 0 0 0 0 1
ASC "a" = #61   0 1 1 0 0 0 0 1
```

Line 150 stores this new value into the lower half of screen memory using absolute indexed addressing. The X register is incremented (line 160) to point to the next location to be moved; the Y register is decremented (line 170), and if not at zero, the program loops back to line 130 for the next byte. When the Y register reaches zero the machine code returns control to BASIC (line 190).

The instructions associated with absolute indexed addressing are shown in Table 10.3.

Table 10.3

Absolute indexed addressing instructions

*ADC	Add with carry	**LDX	Load X register
*AND	Logical AND		Load Y register
ASL	Arithmetic shift left	LSR	Logical shift right
*CMP	Compare memory	*ORA	Logical OR
DEC	Decrement memory	ROL	Rotate left
*EOR	Logical exclusive OR	ROR	Rotate right
INC	Increment memory	*SBC	Subtract with carry
*LDA	Load accumulator	*STA	Store accumulator

Unmarked commands are available with X register as index only. Commands marked * may use either register, whereas the one marked ** can only use the Y register.

INDIRECT ADDRESSING

Indirect addressing allows us to read or write to a memory address which is not known at the time of writing the program! Crazy! Not really, the program itself may calculate the actual address to be handled. Alternatively, a program may contain within it several tables of data which are all to be manipulated in a similar manner. Rather than writing a separate routine for each, a general purpose one can be developed, with the address operand being '*seeded*' on each occasion the routine is called.

Indirect addressing's beauty is that it enables the whole of the Oric's memory map to be accessed with a single two byte instruction. To distinguish indirect addressing from other addressing modes, the operands must be enclosed in brackets.

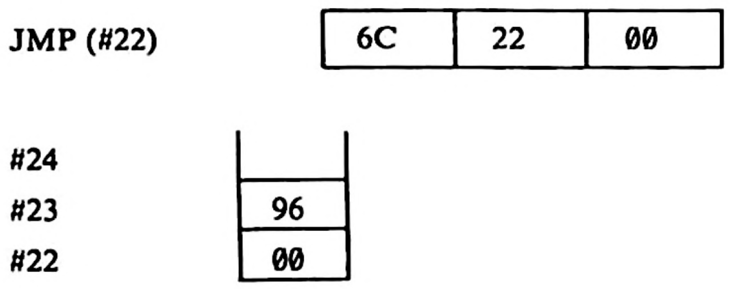
Pure indexed addressing is only available to one instruction—the jump instruction—which is mnemonically represented by JMP. We will look at JMP's function in more detail during the course of Chapter 13, but suffice to say for now that it is the 6502's equivalent of BASIC's GOTO statement. (Though it does of course jump to an address rather than a line number.)

A typical indirect jump instruction takes the form:

```
JMP (#22)
```

The address specified in the instruction is not the address jumped to, but is the address of the location where the jump address is stored. In other words, don't jump here but to the address stored here!

Operation:



From the operational example we can see that location #22 contains the low byte of the address, and location #23 the high byte. These two locations, which act as temporary stores for the address, are known as a *vector*. Executing JMP (#22) in this instance will cause the program to jump to the location #9600.

Program 12 illustrates the use of an indirect jump. The machine code for this program is stored in two different areas of memory identified by VECT and CODE (lines 20 and 30). The machine code at VECT (lines 140 to 180) is responsible for setting up the vector itself. It uses the two zero page locations #70 and #71, and in effect performs a machine code DOKE to place #9600 into these locations. Note that, as always with the 6502, the low byte is stored first. Line 180 then performs the indirect jump to #9600 via #70. The machine code at #9600 places a row of asterisks across the top of the TEXT mode screen.

Program 12

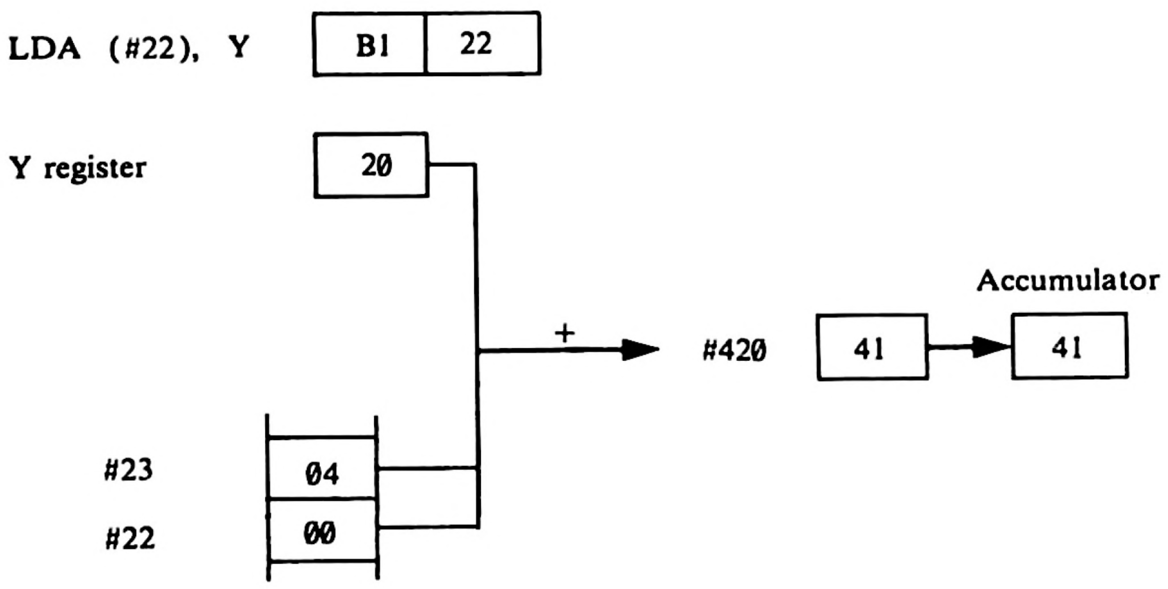
```
10  REM * * INDIRECT JUMPING * *
20  VECT = #400
30  CODE = #9600
40  FOR LOOP = 0 TO 10
50      READ BYTE
60      POKE VECT + LOOP, BYTE
70  NEXT LOOP
80  FOR LOOP = 0 TO 10
```

```
90        READ BYTE
100       POKE CODE + LOOP, BYTE
110    NEXT LOOP
120
130    REM * * M/C DATA * *
135    REM * * DATA FOR VECT * *
140    DATA #A9, #00                 :   REM LDA @0
150    DATA #85, #70                 :   REM STA #70
160    DATA #A9, #96                 :   REM LDA @#96
170    DATA #85, #71                 :   REM STA #71
180    DATA #6C, #70, #00            :   REM JMP (#70)
190    REM * * DATA FOR CODE * *
200    DATA #A2, #28                 :   REM LDX @#28
210    DATA #A9, #2A                 :   REM LDA @ "*"
220    DATA #9D, #A7, #BB            :   REM STA #BBA7, X
230    DATA #CA                      :   REM DEX
240    DATA #D0, #FA                 :   REM BNE -6
250    DATA #60                      :   REM RTS
260
270    REM * * SET UP & EXECUTE * *
280    CLS
290    CALL (VECT)
```

POST-INDEXED INDIRECT ADDRESSING

Post-indexed addressing is a little like absolute indexed addressing, but in this case, the base address is stored in a zero page vector which is accessed indirectly.

Operation:



In the example above, the base address is stored in the vector at #22 and #23. The contents of the Y register (#20) are added to the address in the vector (#400) to give the actual address (#420) of the data. It should be obvious that this form of indirect addressing allows access to a 256 byte range of locations. In the case above, any location from #400 to #4FF is available by setting the Y register accordingly.

Program 13 uses post-indexed indirect addressing to move a line of screen memory from the upper to the lower half of the screen.

Program 13

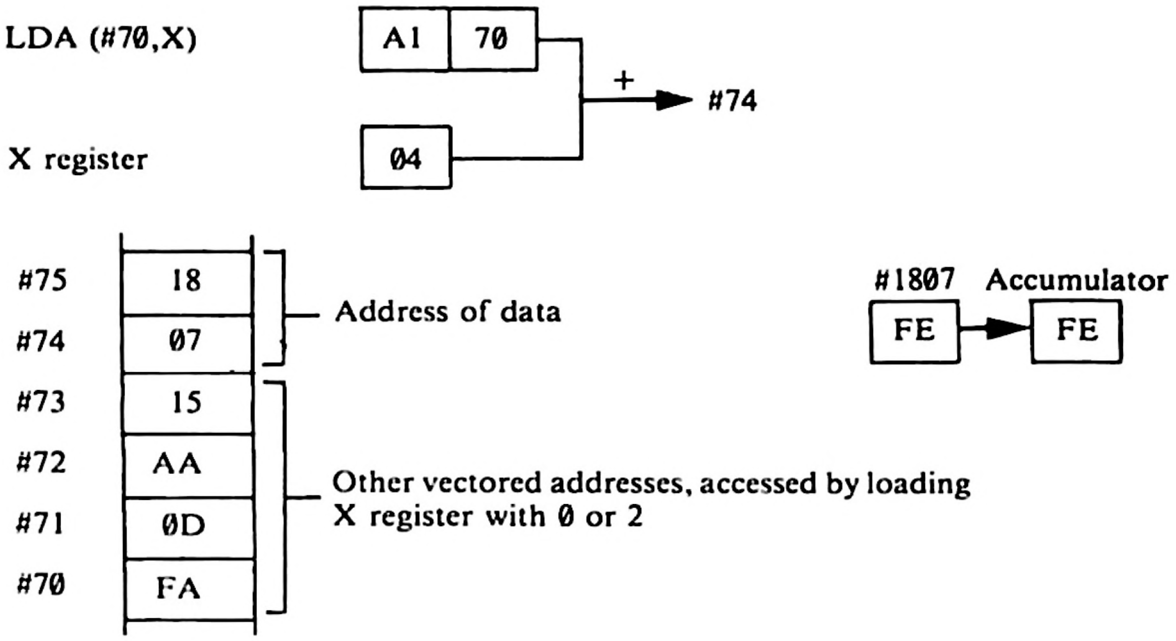
```
 10  REM * * INDIRECT ADDRESSING * *
 20  CODE = #400
 30  FOR LOOP = 0 TO 12
 40     READ BYTE
 50     POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 90
100  DATA #A0, #00          :  REM LDY @#00
110  DATA #A2, #27          :  REM LDX @#27
120  DATA #B1, #70          :  REM LDA (#70), Y
130  DATA #91, #72          :  REM STA (#72), Y
140  DATA #C8               :  REM INY
150  DATA #CA               :  REM DEX
160  DATA #D0, #F8          :  REM BNE -9
170  DATA #60               :  REM RTS
180
190  REM * * SET UP & EXECUTE * *
200  CLS
210  DOKE #70, #BBAA        :  REM screentop
220  DOKE #72, #BCEA        :  REM screenbottom
230  PRINT "INDIRECT ADDRESSING"
240  CALL (CODE)
```

When RUN the two addresses, screentop and screenbottom, are DOKEd into the four zero page locations starting from #70. Once the screen is cleared, the title is printed at the top of the screen (line 230). On calling the machine code (line 240) the Y register is initialized to zero (line 100) and the X register loaded with the number of bytes to be moved (line 110). The byte at the address obtained when the contents of the Y register are added to the address stored in the vector corresponding to screentop, is loaded into the accumulator using indirect addressing (line 120). This byte is then stored at the address given by adding the Y register's contents to the address held in the vector corresponding to screenbottom (line 130). The Y register is then incremented (line 140) and the X register decremented and checked for zero (lines 150 and 160).

PRE-INDEXED ABSOLUTE ADDRESSING

This addressing mode is used if we wish to indirectly access a whole series of absolute addresses which are stored in zero page.

Operation:



Here the contents of the X register (#04) are added to the zero page address (#70) to give the vector address (#74). The two bytes here are then interpreted as the actual address of the data (#1807). Setting the X register to #02 gives indirect access to the vector address #15AA.

A list of instructions which can be used with pre- and post-indexed addressing is shown in Table 10.4.

Table 10.4

Pre- and post-indexed indirect addressing instructions	
ADC	Add with carry
AND	Logical AND
CMP	Compare memory
EOR	Logical exclusive OR
LDA	Load accumulator
ORA	Logical OR
SBC	Subtract with carry
STA	Store accumulator

IMPLIED AND RELATIVE ADDRESSING

Two other modes of addressing are available with the 6502 namely, *implied* addressing and *relative* addressing. We will be dealing with both of these addressing modes during the course of the next few chapters.

LOOK-UP TABLES

Look-up tables can provide an easy way out of what might otherwise be a complicated piece of machine code—such as calculating the square roots of equations. As an example, let's develop a machine code program to convert degrees Centigrade into degrees Fahrenheit. The formula for this is:

°F = 1.8 (°C) + 32

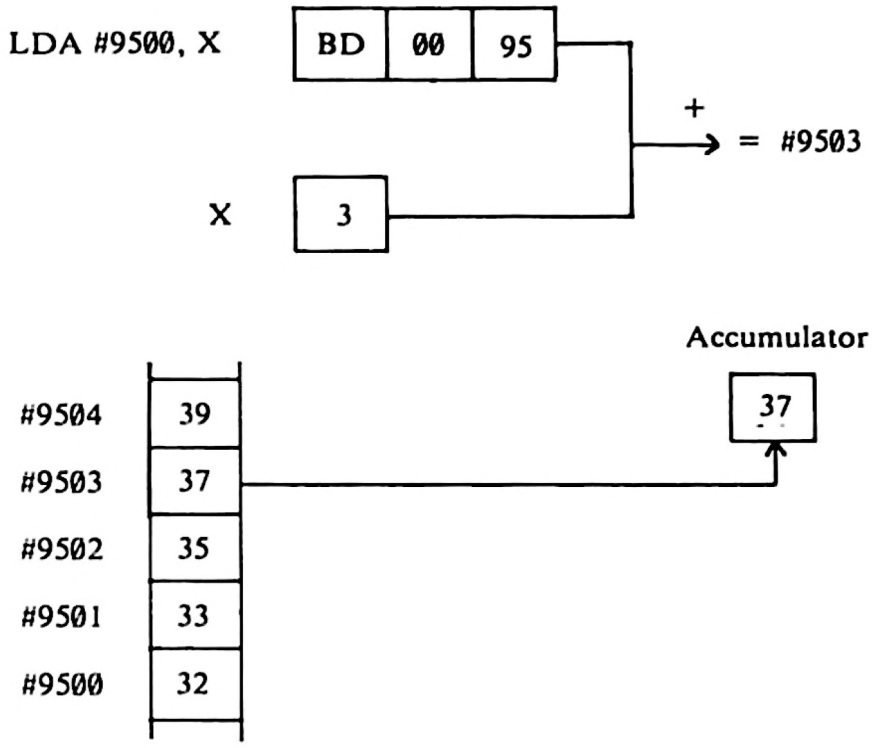
As can be seen, a decimal multiplication (1.8 × C) is required, followed by an addition. This is both time consuming and somewhat painful to the grey matter! By providing the relative Fahrenheit values precalculated in a table, the Centigrade value can be used as the index to the table to obtain the equivalent Fahrenheit value.

Program 14

```
10  REM * * LOOK-UP TABLE * *
20  REM * * 'HIMEM #9600' * *
30  CODE = #9600
40     FOR LOOP = 0 TO 9
50     READ BYTE
60     POKE CODE + LOOP, BYTE
70  NEXT LOOP
80  REM * * set up C to F table * *
85  TABLE = #9500
90  FOR N = 0 TO 100
100   DEGREE = (1.8 * N) + 32
110   POKE TABLE + N, DEGREE
120 NEXT N
130
140 REM * * M/C DATA * *
150 DATA #AE, #00, #40          :  REM LDX #400
160 DATA #BD, #00, #95          :  REM LDA #9500, X
170 DATA #8D, #00, #04          :  REM STA #400
180 DATA #60                    :  REM RTS
190
200 REM * * SET UP & EXECUTE * *
205 CLS
210 REPEAT
220    INPUT "Centigrade value :";C
230    POKE #400, C
240    CALL (CODE)
250    PRINT "Fahrenheit value :";
260    PRINT PEEK (#400)
270 UNTIL C > 100
```

Lines 80 to 120 calculate the equivalent Fahrenheit values for the Centigrade temperatures in the range 0° to 100°, and poke them into a table which has its base at #9500. The example shows how the Centigrade value is used as an index to obtain the Fahrenheit value.

Example: Convert 3°C to Fahrenheit.



Therefore 3°C is equivalent to 37°F.

11 Stacks of Fun

THE STACK

The stack is perhaps one of the more difficult aspects of the 65Ø2 to understand, however it is well worth the time mastering as it lends itself to more efficient programming. Because of its importance the whole of *Page* #Ø1 (that is memory locations #1ØØ through to #1FF) is given over to its operation.

The stack is used as a temporary store for data and memory addresses that need to be remembered for use sometime later on during the program. For most purposes its operation is transparent to us. For example, when, during the course of a BASIC program a GOSUB is executed, the address of the next BASIC command or statement after it is placed onto the stack, so that the program knows where to return to on completion of the subroutine. The process of placing values onto the stack is known as *pushing*, whilst retrieving the data is called *pulling*.

The stack has one important feature which must be understood—it is a *last in, first out* (LIFO) structure. What this means is that the last byte pushed onto the stack must be the first byte pulled from it.

A useful analogy to draw here is that of a train yard. Consider a small spur line, onto which trucks 1, 2 and 3 are pushed (see Figure 11.1). The first truck onto the line (truck 1) is at the very end of the line, truck 2, the second onto the line is in the middle, and the last truck (truck 3) is nearest the points.

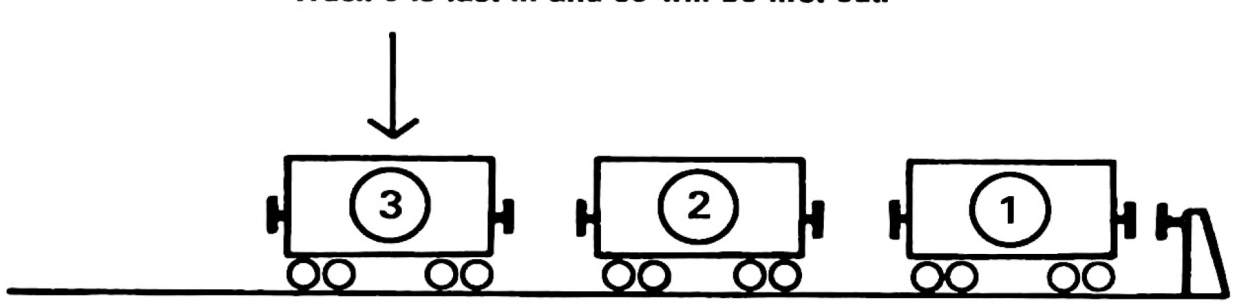


Figure 11.1 The stack—LIFO.

It should now be fairly obvious that the first truck to be pulled off the spur must be the last truck pushed onto it, that is, truck 3. Truck 2 will be the next to be pulled from the line, and the first truck in will be the last one out.

To help us keep track of our position on the stack, there is a further 65Ø2 register called the *Stack Pointer.* Because the stack is hardware item of the 65Ø2, that is, it is actually 'wired' into it, the 'page number' of the stack (#Ø1) can be omitted from the address, and the Stack Pointer just points to the next free position in the stack.

When the Oric is switched on (or a BREAK is performed) the Stack Pointer is loaded with the value #FF—it points to the top of the stack—this means that the stack grows down the memory map rather than up as may be expected. Each time an item is pushed the Stack Pointer is decremented, and conversely, it is incremented when the stack is pulled.

STACK INSTRUCTIONS FOR SAVING DATA

The 65Ø2 has four instructions that allow the accumulator and Status register to be pushed and pulled. They are:

PHA	Push accumulator onto stack
PLA	Pull accumulator from stack
PHP	Push Status register onto stack
PLP	Pull Status register from stack

All four instructions use *implied* addressing and occupy only a single byte of memory.

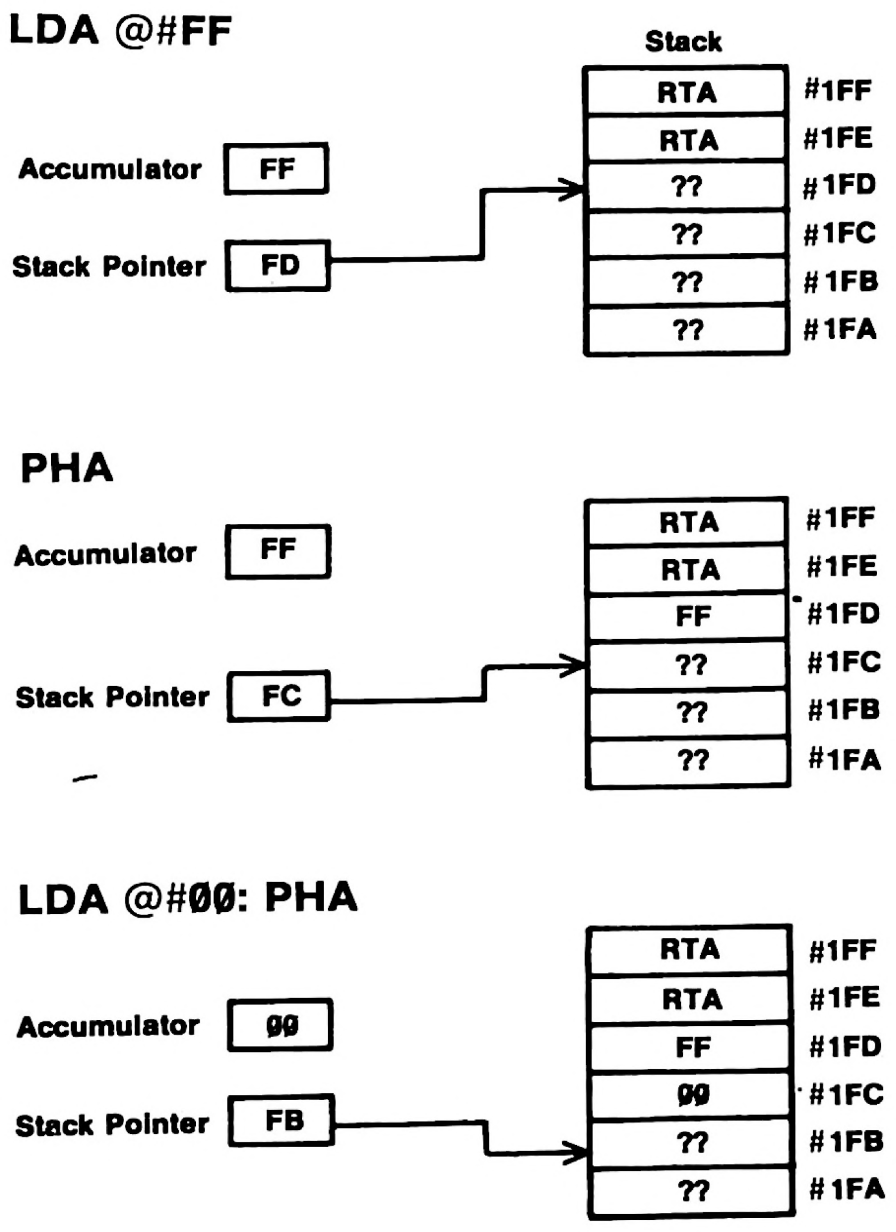


Figure 11.2 Pushing items on to the stack.

The PHA and PHP instructions work in a similar manner, but on different registers. In both cases the source register remains unaltered by the instruction. Again PLA and PLP are similar in operation, but PLA conditions only the Negative and Zero flags, while PLP of course conditions all the flags.

Consider the following sequence of instructions:

```
TWOPUSH  LDA @#FF          \ place #FF in accumulator
         PHA               \ push onto stack
         LDA @#00          \ place #00 in accumulator
         PHA               \ push onto stack
```

Figure 11.2 shows exactly what happens as this program is executed. The Stack Pointer (SP) at the start contains #FD and points to the next free location in the stack. The first two stack locations #FF and #FE hold the two byte return address (RTA) to which the machine code will eventually pass control. (This may be the address of the next BASIC instruction if a call to machine code has been made.) The subsequent stack locations are at present undefined and are therefore represented as ??.

After the accumulator has been loaded with #FF it is copied onto the stack by PHA. Note that the accumulator's contents are not affected by this operation. Once it has been pushed onto the stack, the Stack Pointer's value is decremented by one to point to the next free location in the stack (#FC).

The accumulator is then loaded with #00 and this is pushed on to the stack at location #FC. The Stack Pointer is again decremented to the next free location (#FB).

To remove these items from the stack the following could be used:

```
TWOPULL  PLA               \ get #00 from stack
         STA #70           \ save it somewhere
         PLA               \ get #FF from stack
         STA #71           \ save it as well
```

Figure 11.3 illustrates what happens in this case. The first PLA will pull from the stack into the accumulator the last item pushed onto it, which in this example is #00. The Stack Pointer is incremented, this time to point to the new 'next free location', #FC. As you can see from the diagram the stack contents are not altered, but the #00 will be overwritten if a further item is now pushed. The STA #70 saves the accumulator value somewhere in memory so that it is not destroyed by the next PLA. This PLA restores the value #FF into the accumulator, and again increments the Stack Pointer.

One thing should now be apparent—it is very important to remember the order in which items are pushed onto the stack, as they *must* be pulled in exactly the reverse order. If this process is not strictly adhered to then errors will certainly result, and could even cause your program to crash or hang-up!

The following program shows how the stack can be used to save the contents of the various registers to be printed later. This is particularly useful for debugging those awkward programs that just will not work.

```
REGSAVE  PHP               \ save Status register
         PHA               \ save accumulator
         TXA               \ transfer X into accumulator
         PHA               \ save accumulator (X)
         TYA               \ transfer Y into accumulator
         PHA               \ save accumulator (Y)
```

It is important to save the registers in the order shown. The Status register should be saved first so that it will not be altered by the subsequent transfer instructions which could affect the Negative and Zero flags, and the accumulator must be saved before its value is destroyed by either of the index register transfer operations.

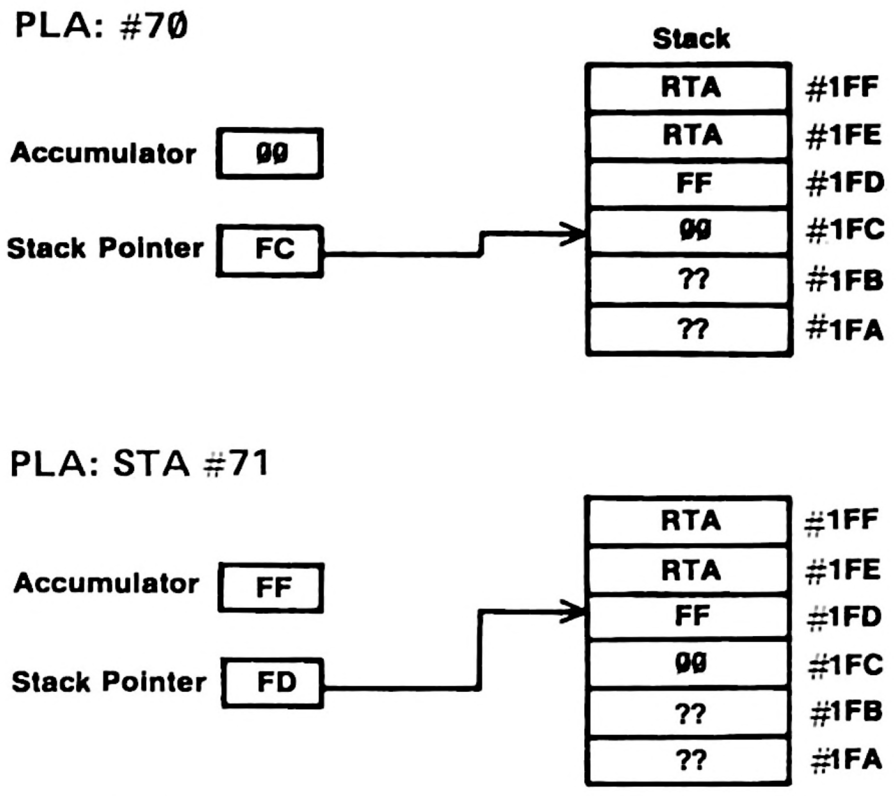


Figure 11.3 Pulling items from the stack.

If the registers' values had been saved to preserve them while another portion of the program was operating, we could retrieve them with:

```
PLA                         \ pull accumulator (Y)
TAY                         \ and transfer to Y register
PLA                         \ pull accumulator (X)
TAX                         \ and transfer to X register
PLA                         \ pull accumulator
PLP                         \ pull Status register
```

There are two final stack associated instructions:

TSX	Transfer Stack Pointer to X register
TXS	Transfer X register to Stack Pointer

We shall see how and why the stack is used to save addresses in Chapter 13.

12 Looping

LOOPS

Loops allow sections of programs to be repeated over and over again. For example, in BASIC we could print ten exclamation marks using a FOR . . . NEXT loop like this:

```
10  FOR NUM = 0 TO 9
20  PRINT "!" ;
30  NEXT NUM
```

In line 10 a counter called NUM is declared and initially set to zero. Line 20 prints the exclamation mark, and line 30 checks the present value of NUM to see if it has reached its maximum limit. If it has not, the program adds one to NUM and branches back to print the next exclamation mark.

To implement this type of loop in machine code we need to know how to control and use the three topics identified above; namely counters, comparisons and branches.

COUNTERS

It is usual to use index registers as counters, because they have their own increment and decrement instructions.

INX	Increment X register	X = X + 1
INY	Increment Y register	Y = Y + 1
DEX	Decrement X register	X = X − 1
DEY	Decrement Y register	Y = Y − 1

All these instructions can affect the Negative and Zero flags. The Negative flag is set if the most significant bit of the register is set following an increment or decrement instruction—otherwise it will be cleared. The Zero flag will only be set if any of the instructions cause the register concerned to contain zero.

Note that incrementing a register which contains #FF will reset that register to #00, will clear the Negative flag (N = 0) and will set the Zero flag (Z = 1). Conversely, decrementing a register holding #00 will reset its value to #FF, set the Negative flag and clear the Zero flag.

There are two other increment and decrement instructions:

INC	Increment memory
DEC	Decrement memory

These instructions allow the values in memory locations to be adjusted by one, for example:

```
INC #70                          \ add 1 to location #70
DEC #420                         \ subtract 1 from location #420
```

Both instructions condition the Negative and Zero flags as described earlier.

Program 15 shows how these instructions can be used, in this case to print 'ABC' on the screen.

Program 15

```
 10  REM * * INCREMENTING A REGISTER * *
 20  CODE = #400
 30  FOR LOOP = 0 TO 11
 40     READ BYTE
 50     POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 90  DATA #A9, #41            : REM LDA @"A"
100  DATA #85, #70            : REM STA #70
110  DATA #AA                 : REM TAX
120  DATA #E8                 : REM INX
130  DATA #86, #71            : REM STX #71
140  DATA #E8                 : REM INX
150  DATA #86, #72            : REM STX #72
160  DATA #60                 : REM RTS
170
180  REM * * SET UP & EXECUTE * *
190  CLS
195  CALL (CODE)
200  FOR LOC = 0 TO 2
210     NUM = PEEK(#70 + LOC)
220     PRINT CHR$(NUM)
230  NEXT LOC
```

COMPARISONS

There are three compare instructions:

CMP	Compare accumulator
CPX	Compare X register
CPY	Compare Y register

The contents of any register can be compared with the contents of a specified memory location, or as is often the case, the value immediately following the mnemonic. The values being tested remain unaltered. Depending on the result of the comparison, the Negative, Zero and Carry flags are conditioned. How are these flags conditioned? Well, the first thing the 65Ø2 does is set the Carry flag (C = 1). It then subtracts the specified value from the contents of the register. If the value is less than, or equal to the register contents, the Carry flag remains set. If the two values are equal the Zero flag is also set. If the Carry flag has been cleared, it means that the value was greater than the register contents, and a borrow occurred during the subtraction. The Negative flag is generally (but not always) set when this occurs—this is only really valid for two's complement compares. Table 12.1 summarizes these tests.

Examples of compare instructions include:

```
CMP @#41

CPY #42Ø, X

CPX #96ØØ
```

Table 12.1

Test	Flags		
	C	Z	N
Register less than data	Ø	Ø	1
Register equal to data	1	1	Ø
Register greater than data	1	Ø	Ø

BRANCHES

Depending on the result of a comparison, the program will need either to branch back to repeat the loop, branch to another point in the program, or just simply continue. This type of branching is called *conditional branching*, and eight instructions enable various conditions to be evaluated. The branch instructions are:

BNE	Branch if not equal	Z = Ø
BEQ	Branch if equal	Z = 1
BCC	Branch if Carry clear	C = Ø
BCS	Branch if Carry set	C = 1
BPL	Branch if plus	N = Ø
BMI	Branch if minus	N = 1
BVC	Branch if overflow clear	V = Ø
BVS	Branch if overflow set	V = 1

Let's now rewrite the BASIC program to print ten exclamation marks (see page 54) in machine code.

Program 16

```
1Ø  REM * * 1Ø ! MARKS * *
2Ø  CODE = #4ØØ
3Ø   FOR LOOP = Ø TO 12
4Ø      READ BYTE
5Ø      POKE CODE + LOOP, BYTE
```

```
 60   NEXT LOOP
 70
 80   REM * * M/C DATA * *
 90   DATA #A2, #00                : REM LDX @#00
100   DATA #A9, #21                : REM LDA @ '!'
110   DATA #9D, #AA, #BB           : REM STA #BBAA, X
120   DATA #E8                     : REM INX
130   DATA #E0, #0A                : REM CPX @10
140   DATA #D0, #F8                : REM BNE -8   ?
150   DATA #60                     : REM RTS
160
170   REM * * SET UP & EXECUTE * *
180   CLS
190   CALL (CODE)
```

Lines 90 and 100 initialize the X register and place the ASCII code for an exclamation mark in the accumulator. After the '!' is stored in screen memory (line 110) the X register is incremented (line 120) and line 130 tests to see if all the exclamation marks have been printed. If the comparison fails (X <> 10) the Zero flag will remain clear and the BNE of line 140 will be executed, causing the program to branch back to line 110. If the comparison is true, the Zero flag is set and the RTS of line 150 executed.

We can make this program more efficient. Whenever a register is used as a loop counter, and *only* as a loop counter, it is best to write the code so that the register counts down rather than up. Why? Well, you may recall from Chapter 6 that when a register is decremented such that it holds zero the Zero flag is set. Using this principle, the CPX @ 10 can be completely removed, and the program rewritten as shown in Program 17.

Program 17

```
 10   REM * * DOWN COUNT * *
 20   CODE = #400
 30   FOR LOOP = 0 TO 12
 40      READ BYTE
 50      POKE CODE + LOOP, BYTE
 60   NEXT LOOP
 70
 80   REM * * M/C DATA * *
 90   DATA #A2, #0A                : REM LDX @10
100   DATA #A9, #21                : REM LDA @'!'
110   DATA #9D, #AA, #BB           : REM STA #BBAA, X
120   DATA #CA                     : REM DEX
140   DATA #D0, #FA                : REM BNE -6
150   DATA #60                     : REM RTS
160
170   REM * * SET UP & EXECUTE * *
```

```
180   CLS
190   CALL (CODE)
```

If you look closely at the listing you will notice that the BNE opcode is followed by a single byte and not an address as you may have expected. This is known as the *displacement* and the mode of addressing is called *relative*. The operand, in this case #FA, tells the processor that a backward branch of 6 bytes is required.

To distinguish branches backwards from branches forwards signed binary is used. A negative value means a backward branch while a positive number indicates a forward branch. Obviously, it is important to know how to calculate these displacements—so let's try it.

Before sitting down in front of your Oric it is always best to sit down and write your machine code program. While it is perfectly feasible to write it at the keyboard, this nearly always leads to errors in the coding (I speak from experience!).

To make it clear just where loops are branching to and from, you can use *labels*. With Program 17, for example, you can write the mnemonics using the label LOOP to mark the destination of the branch, as shown in Table 12.1.

Table 12.1

Label	Mnemonics	Comments
START		Code begins
	LDX @10	Loop counter
	LDA @'!'	ASCII Code for !
LOOP		Branch destination
	STA #BBAA, X	Store in screen memory
	DEX	X = X − 1
	BNE LOOP	Continue until X = 0
	RTS	All done!

Once this is done, we can look up the relevant machine code and insert it into the table. To calculate the branch displacement, just count the number of bytes from the displacement byte itself *back* to the label LOOP.

```
LOOP
          STA #BBAA, X      3 bytes
          DEX               1 byte
          BNE LOOP          2 bytes
```

This gives a total displacement of 6 bytes. Note that the relative displacement is included in the count (although it is not known at the time of calculation) because the Program Counter will be pointing to the instruction following it.

To convert this displacement to its backward branch signed binary form, obtain the two's complement value (see Chapter 3 if you need some refreshing on just how to do this).

```
  0 0 0 0 0 1 1 0       (6)
  1 1 1 1 1 0 0 1
+               1
 ----------------
  1 1 1 1 1 0 1 0       (−6 = #FA)
```

Since all branches are two bytes long, effective displacements of −126 bytes (−128 + 2) and +129 bytes (127 + 2) are possible.

To demonstrate the use of a forward branch enter and RUN the following program, which will display a Y if location #400 contains a '0' or an N otherwise.

Program 18

```
 10  REM * * FORWARD BRANCHING * *
 20  REM * * SET HIMEM #9600 * *
 30  CODE = #9600
 40  FOR LOOP = 0 TO 16
 50     POKE CODE + LOOP, BYTE
 60     READ BYTE
 70  NEXT LOOP
 80
 90  REM * * M/C DATA * *
100  DATA #AD, #00, #04          :  REM LDA #400
110  DATA #F0, #06               :  REM BEQ +6
120  DATA #A9, #4E               :  REM LDA @'N'
130  DATA #8D, #AA, #BB          :  REM STA #BBAA
140  DATA #60                    :  REM RTS
150  DATA #A9, #59               :  REM LDA @'Y'
160  DATA #8D, #AA, #BB          :  REM STA #BBAA
170  DATA #60                    :  REM RTS
180
190  REM * * SET UP & EXECUTE * *
200  CLS
210  CALL (CODE)
```

When Program 18 is RUN, the contents of location #400 are loaded into the accumulator (line 100). If the contents are #00, the Zero flag is set and the BEQ of line 110 performed, thus jumping over the bytes responsible for storing the 'N' into screen memory. Because the branch is in a forward direction, a positive value (that's one less than #80) is used as the displacement. The labelled mnemonics I used to calculate the displacement are shown in Table 12.2.

Table 12.2

Label	Mnemonics	Comments
START		At #9600
	LDA #400	Get byte
	BEQ ZERO	Branch if Z = 0
	LDA @'N'	ASCII code for N
	STA #BBAA	Store in screen memory
	RTS	Back to BASIC
ZERO	LDY @'Y'	ASCII code for Y
	STA #BBAA	Store in screen memory
	RTS	Back to BASIC

Counting the bytes from the byte after the branch to the label ZERO we get:

```
          BEQ ZERO
          LDA @'N'       2 bytes
          STA #BBAA      3 bytes
          RTS            1 byte
    ZERO
```

a total of 6 bytes. Appendix 4 contains two tables which can be used to convert the byte count into the correct displacement rather than calculating it by hand in this way.

MEMORY COUNTERS

Invariably programs that operate on absolute addresses will require routines that are capable of incrementing or decrementing these double byte values. A typical case being a program using post-indexed indirect addressing that needs to sequentially access a whole range of consecutive memory locations. The following two programs show how this can be done. First, incrementing memory addresses.

Program 19

```
 10  REM * * INCREMENTING MEMORY * *
 20  REM * * SET HIMEM #9600 * *
 30  MCODE = #9600
 40  COUNTER = #400
 50  DOKE COUNTER, 0
 60  FOR LOOP = 0 TO 8
 70    READ BYTE
 80    POKE MCODE + LOOP, BYTE
 90  NEXT BYTE
100
110  REM * * M/C DATA * *
120  DATA #EE, #00, #04          :  REM INC #400
130  DATA #D0, #03               :  REM BNE +3
140  DATA #EE, #01, #40          :  REM INC #401
150  DATA #60                    :  REM RTS
160
170  REM * * SET UP & EXECUTE * *
180  REPEAT
190    PRINT DEEK (COUNTER)
200    CALL (MCODE)
210    GET A$
220  UNTIL A$ = "S"
```

Here the relative machine code is contained in the lines 120 to 140. Each time the routine is called, the low byte of #400 is incremented. Each time it goes from #FF to #00 the Zero flag is set and the branch (line 130) will not take place—and therefore the high byte of the counter at #401 is incremented.

Decrementing a counter is a little less straightforward.

Program 20

```
10   REM * * DECREMENTING MEMORY * *
20   REM * * SET HIMEM #9600 * *
30   MCODE = #9600
40   COUNTER = #400
50   DOKE COUNTER, #FFFF
60   FOR LOOP = 0 TO 11
70     READ BYTE
80     POKE MCODE + LOOP, BYTE
90   NEXT BYTE
100
110  REM * * M/C DATA * *
120  DATA #AD, #00, #04          :   REM LDA #400
130  DATA #D0, #03               :   REM BNE +3
140  DATA #CE, #01, #04          :   REM DEC #401
150  DATA #CE, #00, #04          :   REM DEC #400
160  DATA #60                    :   REM RTS
170
180  REM * * SET UP & EXECUTE * *
190  REPEAT
200    PRINT DEEK (COUNTER)
210    CALL (MCODE)
220    GET A$
230  UNTIL A$ = "S"
```

The accumulator is first loaded with the low byte of the counter at #400 (line 120), this procedure will condition the Zero flag. If it is set, the low byte of counter must contain #00, and therefore the high byte needs to be decremented (line 140). Otherwise it is skipped and only the low byte is decremented. Either of the index registers could have been used in place of the accumulator in line 120. If all registers are in use the following alternative can be employed:

```
          INC #400
          DEC #400
          BNE LSBDEC
          DEC #401
LSBDEC    DEC #400
```

The process of first incrementing and then decrementing the low byte of #400 will condition the Zero flag in the same manner as a *load* instruction.

13 Subroutines and Jumps

SUBROUTINES

If you are familiar with BASIC's GOSUB and RETURN statements you should have little difficulty understanding the two assembler equivalents:

JSR	Jump save return
RTS	Return from subroutine

If you are not familiar, I will explain.

Quite often during the course of writing a program you will find that a specific operation must be performed more than once, perhaps several times. Rather than typing in the same group of mnemonics on every occasion, which is both time consuming and increases the programs' length, they can be entered once, out of the way of the main program flow, and called when required. Not every piece of repetitive assembler warrants being coded into a subroutine, however. For example:

```
INX : DEY : STA #22
```

is quite common (or something very similar) however, when assembled it only occupies four bytes of memory, which is the same memory requirement as a JSR . . . RTS call. Nothing is to be gained by introducing a subroutine here then, in fact, it will actually slow the program operation down by a few millionths of a second! On the other hand:

```
CLC : LDA #22 : ADC @#12 : STA #23
```

might well warrant its own subroutine call as, if absolute addressing is employed, it may be up to ten bytes in length.

Let's now look at a short program which employs several subroutine calls.

Program 21

```
10  REM * * SUBROUTINE DEMO* *
20  REM * * SET HIMEM #9600 * *
30  CODE = #9600
40  SUBR = #400
50  REM * * set up subroutine * *
60  FOR LOOP = 0 TO 3
70    READ BYTE
80    POKE SUBR + LOOP, BYTE
```

```
 90  NEXT LOOP
100  REM * * set up main m/code * *
110  FOR LOOP = 0 TO 15
120    READ BYTE
130    POKE CODE + LOOP, BYTE
140  NEXT LOOP
150
160  REM * * M/C DATA * *
170  REM * * subroutine * *
180  DATA #9D, #AA, #BB        : REM STA #BBAA, X    STORE ASCII IN SCREEN
190  DATA #60                  : REM RTS
200  REM * * main m/code * *
210  DATA #A9, #41             : REM LDA @"A"        IN ACC
220  DATA #AB                  : REM TAY             TRANSFER TO Y REG
230  DATA #A2, #00             : REM LDX @00         INITIALISE X REG
240  DATA #20, #00, #04        : REM JSR #400
250  DATA #E8                  : REM INX             INCREMENT X
260  DATA #C8                  : REM INY             INCREMENT Y
270  DATA #98                  : REM TYA             TRANSFER Y TO ACC
280  DATA #E0, #1A             : REM CPX @26         HAS X INCREMENTED 26 TIMES
290  DATA #D0, #F6             : REM BNE -10         IF NOT LOOP BACK [illegible]
300  DATA #60                  : REM RTS
310
320  REM * * SET UP & EXECUTE * *
330  CLS
340  CALL (CODE)
```

The subroutine itself is located in the user machine code area. As the subroutine is only four bytes long it would really be much more efficient to include it in the main body of the program, however, for the purposes of demonstration is has been deliberately kept short. The main program is stored from #9600 above a reset HIMEM.

The program itself uses absolute indexed addressing to print the alphabet across the screen. When called, the ASCII code for the letter 'A' is placed into the accumulator (line 210) and then copied into the Y register using the appropriate transfer instruction (line 220). This is important because we will require the register to increment this ASCII code in order to obtain the codes for the rest of the letters of the alphabet, and of course, the accumulator has no direct increment instruction (though we could have used CLC, ADC @1).

After the indexing X register is initialized to zero (line 230), the first of 26 subroutine calls is performed (line 240). The subroutine, located way down the memory map at #400, is responsible for placing the ASCII character code (and therefore the character) into screen memory (line 180). The RTS opcode (line 190) then returns control back to the calling program—*not* BASIC. The index registers are both incremented (lines 250 and 260), the next ASCII character code is placed into the accumulator (line 270) and the program is repeated a further 25 times (lines 280 and 290).

Now that we have taken a general overview of subroutine calls and their functions it will be useful to see just how they manage to do what they do.

The two instructions JSR and RTS must perform three functions between them. Firstly, the current contents of the Program Counter must be saved so that control may be returned to the calling program at some stage. Secondly, the 65Ø2 must be told to execute the subroutine once it arrives there. Finally, program control must be handed back to the calling program.

The JSR instruction performs the first two requirements. To save the return address it pushes the two byte contents of the Program Counter onto the stack. The Program Counter at this stage will hold the address of the location containing the third byte of the three which constitute the JSR instruction. After pushing the Program Counter onto the stack, the operand specified by JSR is placed into the Program Counter, which effectively transfers control to the subroutine.

Figure 13.1 shows how these operations take place, and in particular, their effect on the stack. At the time the 65Ø2 encounters the JSR instruction the Program Counter is pointing to the second byte of the two byte operand (Figure 13.1a). The microprocessor pushes the contents of the program Counter onto the stack, low byte first (Figure 13.1b), and then copies the subroutine address into the Program Counter (Figure 13.1c).

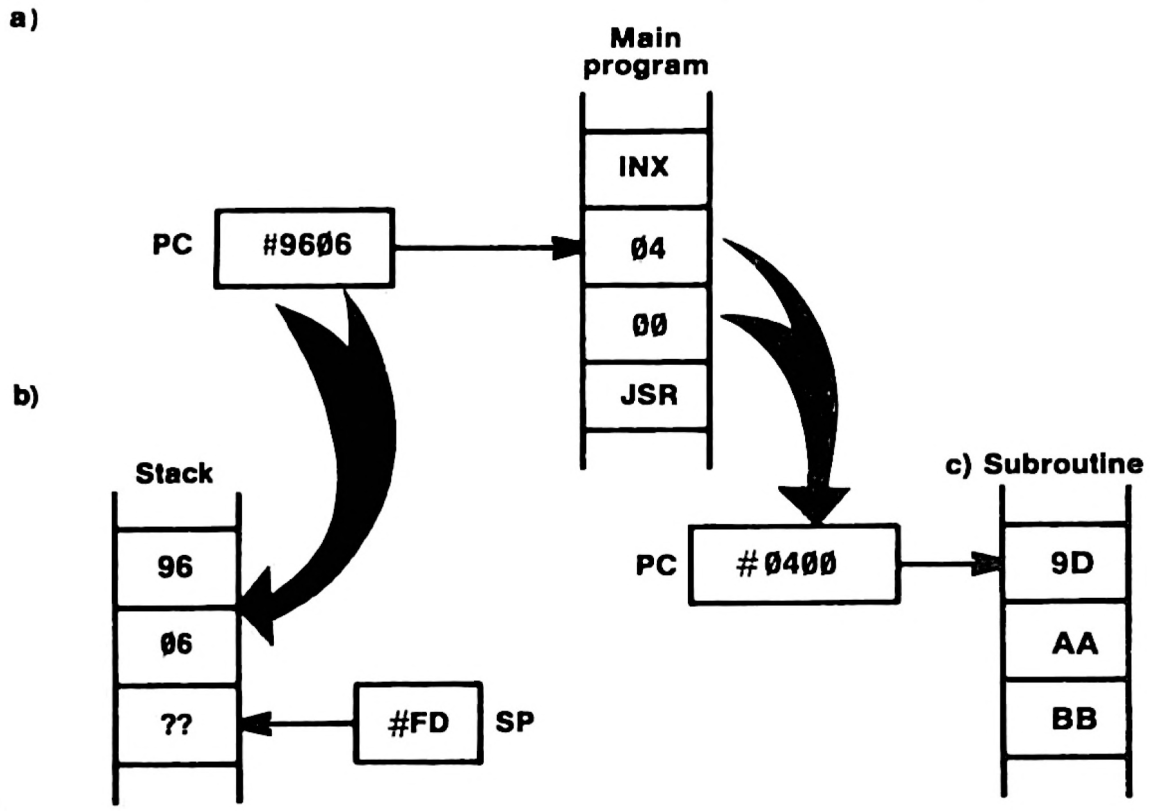


Figure 13.1 Steps taken by a JSR instruction.

When the RTS instruction is encountered at the end of the subroutine, these actions are reversed. The return address is pulled from the stack and incremented by one (Figure 13.2) as it is replaced into the Program Counter, so that it points to the instruction after the original subroutine call.

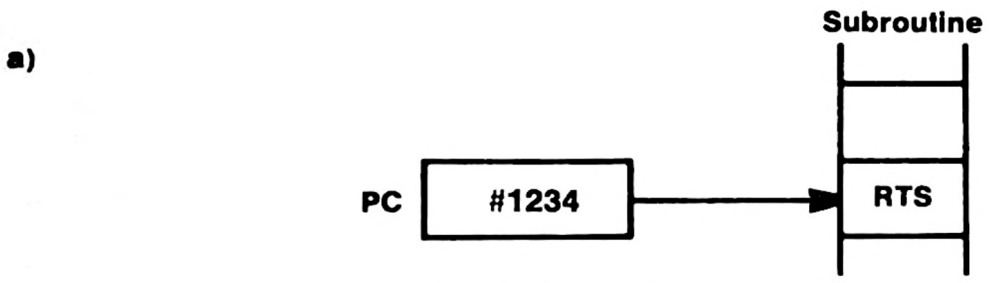
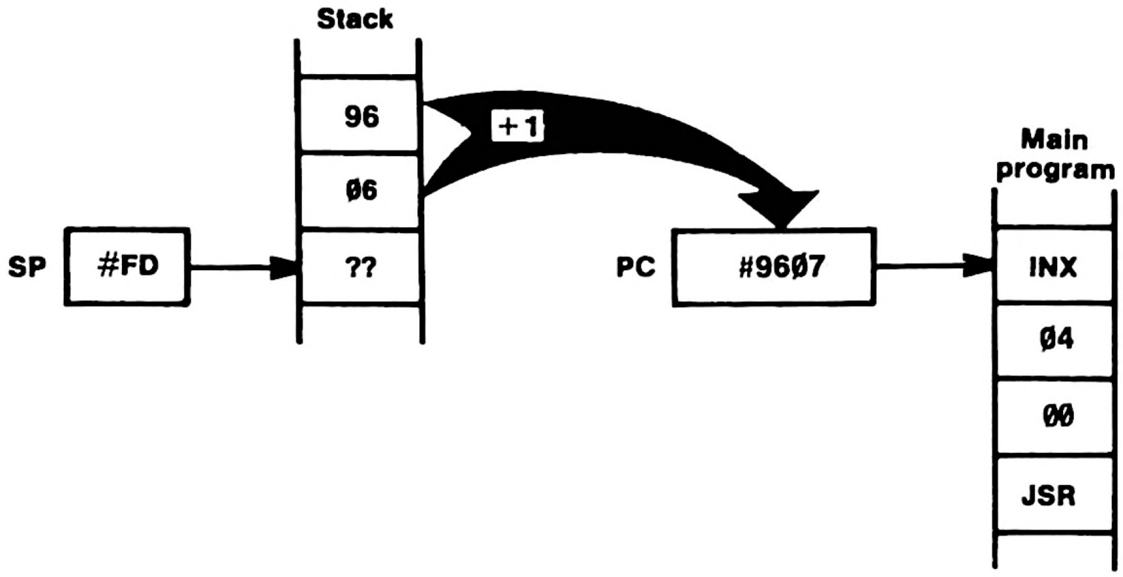


Figure 13.2 Steps taken by an RTS instruction.

b)



c)

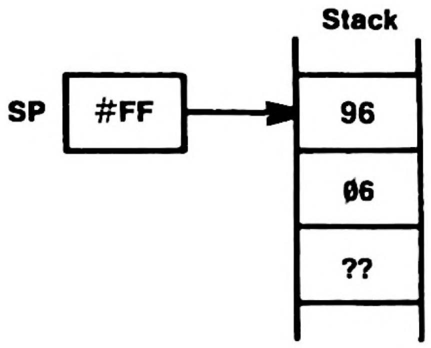


Figure 13.2 (Cont.).

PASSING PARAMETERS

Nine times out of ten a subroutine will require some data to work on, and this will have to be passed into the subroutine by the main program. There are three general ways in which information or parameters can be passed into subroutines, these are:

1. Through registers.
2. Through memory locations.
3. Through the stack.

Let's look at each of these methods in turn.

Through registers

This is quite obviously the simplest method particularly because it can keep the subroutine independent of memory. Because only three registers are available though, only three bytes of information can be conveyed. The registers may themselves contain vital information, so this would need to be saved, possibly on the stack, for future restoration.

Through memory

This is probably the easiest method if numerous bytes are being passed into the subroutine. The best way is to use memory between locations #4ØØ and #42Ø inclusive, because this is reserved for user applications. If the subroutine uses several bytes of memory, a neat way of accessing them is to place the relative start address of the data in the X register, and then use absolute indexed addressing with #400 as the operand as follows:

```
LDX #0
JSR Subroutine
. . .
                    Subroutine    LDA #400, X
                                  . . .
                                  INX
                                  CPX @#10
                                  BNE Subroutine
                                  RTS
```

The disadvantage of using memory locations to pass parameters is that it ties the subroutine to a given area, making it memory dependent. However, on most occasions this does not really matter.

Through the stack

Passing parameters through the stack needs care, since the top of the stack will contain the return address. This method also requires two bytes of memory in which the return address can be saved after pulling it from the stack (though, of course, the index registers could be used). If the stack is used, the subroutine needs to commence with:

```
PLA                           \ pull low byte
STA  ADDR                     \ and save it
PLA                           \ pull high byte
STA  ADDR + 1                 \ and save it
```

The STA instructions can be replaced by TAX and TAY respectively. It is common practice when using the index registers to hold an address, to place the low byte in the X register and the high byte in the Y register.

Once the parameters have been pulled from the stack the return address can be pushed back on to it with,

```
LDA ADDR + 1
PHA
LDA ADDR
PHA
```

Remember, the stack is a LIFO structure, so the bytes need to be accessed and pushed in the reverse order from that in which they were pulled and saved.

If a variable number of parameters is being passed into the subroutine, the actual number can be ascertained each time by evaluating the contents of the Stack Pointer. This can be carried out be transferring its value to the X register with TSX, and incrementing the X register each time the stack is pulled until, say, #FF is reached, indicating the stack is empty. The actual value tested for will depend on whether any other subroutine calls were performed previously—making the current one a nested subroutine. The value #FF is therefore just a hypothetical case and assumes nothing other than that data is present on the stack.

JUMPS

The JMP instruction operates in a similar manner to BASIC's GOTO statement in that it transfers control to another part of the program. In machine code, however, an absolute address is specified rather than a line number (which does not, of course, exist in machine code). The instruction operates simply by placing the two byte address specified after the opcode into the Program Counter, effectively producing a jump. JMP will generally be used to leapfrog over a section of machine code that need not be executed because a test failed. For example:

```
            BCC OVER
            JMP SOMEWHERE
OVER        LDA BYTE
            ASL A
            INX
            DEY
SOMEWHERE   STA TEMP
```

Here, if the Carry flag is clear, the jump instruction will be skipped and the code of OVER executed. If the Carry flag is set the test will fail, the JMP will be encountered and the code of OVER bypassed.

A further use of JMP, the 'indirect jump', was detailed in Chapter 10. As we saw, the address that is actually jumped to is stored in a vector, the address of which is specified in the instruction. JMP (#420) being an example.

14 Shifts and Rotates

Basically, these instructions allow the bits in a single byte to be moved one bit to the left or one bit to the right. There are four instructions available:

ASL	Arithmetic shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right

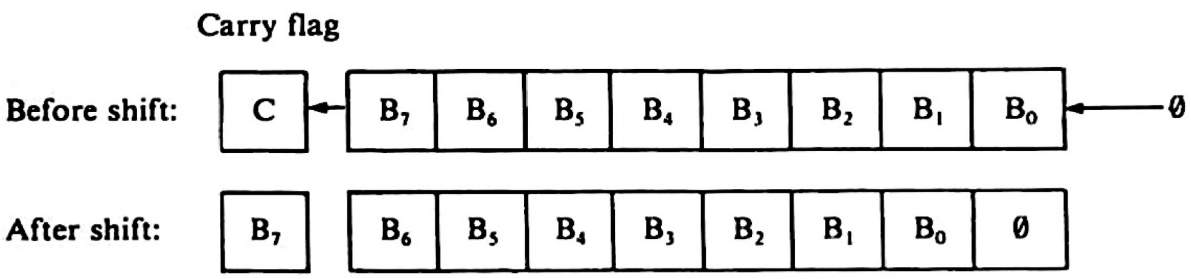
All of these instructions may operate directly on the accumulator, or on a specified memory byte:

```
ASL A          \ arithmetic shift left accumulator
ROL #70        \ rotate left location #70
```

Let's investigate each command in more detail.

ARITHMETIC SHIFT LEFT

ASL moves the contents of the specified byte left by a single bit.



Carry flag
Before shift:
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
0
After shift:
B_7
B_6 B_5 B_4 B_3 B_2 B_1 B_0 0

Bit 7 (B_7) is shifted into the Carry flag, and a '0' takes the place of bit 0 as the rest of the bits are shuffled left. The overall effect is to double the value of the byte in question.

Example:

			C	Accumulator
LDA @#42	\	load accumulator with #42	X	01000010

		C	Accumulator
ASL A	\ shift accumulator left	$\boxed{0}$	$\boxed{10000100}$

The accumulator now holds #84, twice the original value!

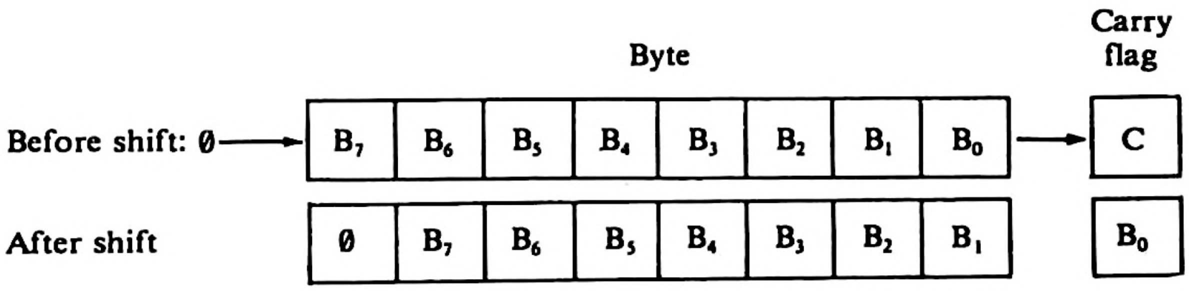
A further example of ASL is given by Program 22 which asks for a number (less than 64), multiplies it by four using ASL A, ASL A, and prints the answer.

Program 22

```
 10  REM * * MULTIPLY BY FOUR * *
 20  CODE = #400
 30  FOR LOOP = 0 TO 8
 40    READ BYTE
 50    POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 90  DATA #AD, #20, #04         :  REM LDA #420
100  DATA #0A                   :  REM ASL A
110  DATA #0A                   :  REM ASL A
120  DATA #8D, #20, #04         :  REM STA #420
130  DATA #60                   :  REM RTS
140
150  REM * * SET UP & EXECUTE * *
160  CLS
170  INPUT "Number to multiply"; NUM
180  POKE #420, NUM
190  CALL (CODE)
200  PRINT "Result : ";
210  PRINT PEEK (#420)
```

LOGICAL SHIFT RIGHT

LSR is similar to ASL except that it moves the bits in the opposite direction, with bit 0 (B_0) jumping into the Carry flag and a 0 following into the spot vacated by bit 7 (B_7).



This instruction could well have been called arithmetic shift right because it effectively divides the byte being shifted by two. For example:

```
                                                     C      Accumulator
LDA @#42    \  load accumulator with #42            [X]    [0 1 0 0 0 0 1 0]

                                                     C
LSR A       \  shift accumulator right              [0]    [0 0 1 0 0 0 0 1]
```

The accumulator now holds #21, half the original value.

Using:

```
LSR A : BCS Elsewhere
```

or,

```
LSR A : BCC Somewhere
```

is a good efficient way of testing bit 0 of the accumulator.

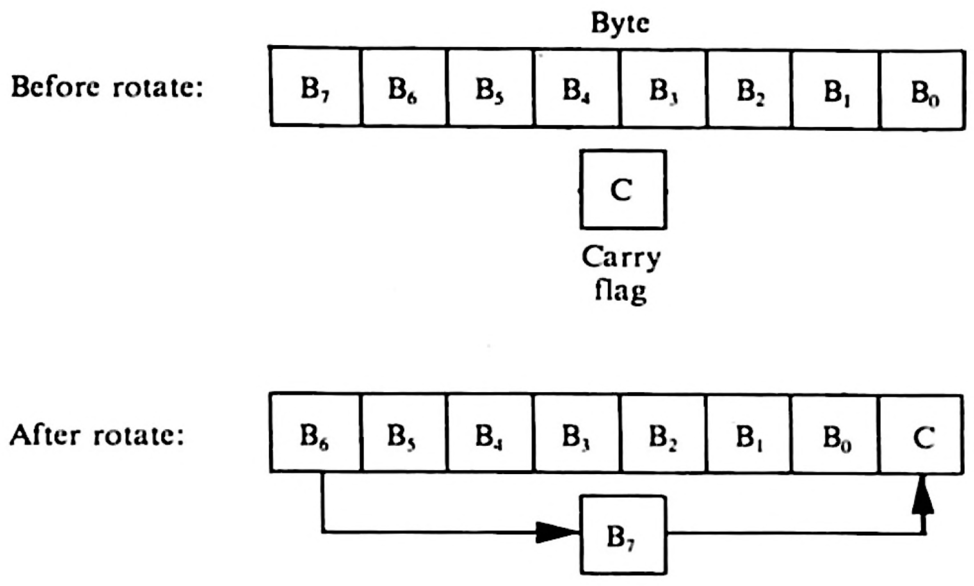
Program 23 tests the condition of bit 0 of an input ASCII character by shifting it into the Carry flag position. If the carry is clear a zero is printed, if set—a one is printed instead.

Program 23

```
 10  REM * * TEST BIT 0 * *
 20  CODE = #400
 30  FOR LOOP = 0 TO 11
 40    READ BYTE
 50    POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 90  DATA #AD, #20, #04        :  REM LDA #420
100  DATA #4A                  :  REM LSR A
110  DATA #A9, #00             :  REM LDA @0
120  DATA #69, #00             :  REM ADC @0
130  DATA #8D, #20, #04        :  REM STA #420
140  DATA #60                  :  REM RTS
150
160  REM * * SET UP & EXECUTE * *
170  CLS
180  INPUT "Test number"; NUM
190  POKE #420, NUM
200  CALL (CODE)
210  PRINT PEEK (#420)
```

ROTATE LEFT

This instruction uses the Carry flag as a ninth bit, rotating the whole byte left one bit in a circular motion, with bit 7 moving into the Carry flag, which in turn moves across to bit Ø.


Byte
Before rotate:
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
C
Carry flag
After rotate:
B_6 B_5 B_4 B_3 B_2 B_1 B_0 C
B_7

ROL provides an easy method of testing any of the four bits constituting the upper nibble of the accumulator. The desired bit is rotated into the bit 7 position, thus setting or clearing the Negative flag as appropriate.

Example: test bit 5 of accumulator.

Accumulator	AF	1 Ø 1 Ø 1 1 1 1	N = 1
Carry flag	Ø		

ROL A / rotate bit 6 into the bit 7 position

Accumulator	5E	Ø 1 Ø 1 1 1 1 Ø	N = Ø
Carry flag	1		

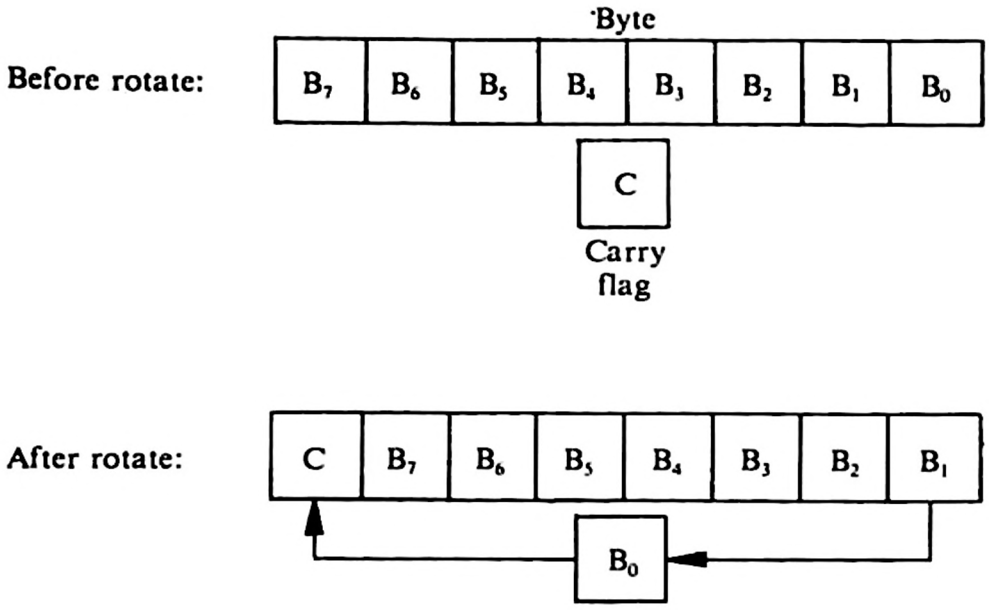
ROL A / rotate bit 5 into the bit 7 position

Accumulator	BC	1 Ø 1 1 1 1 Ø 1	N = 1
Carry flag	Ø		

The Negative flag is now set, indicating that bit 5 of the accumulator was set.

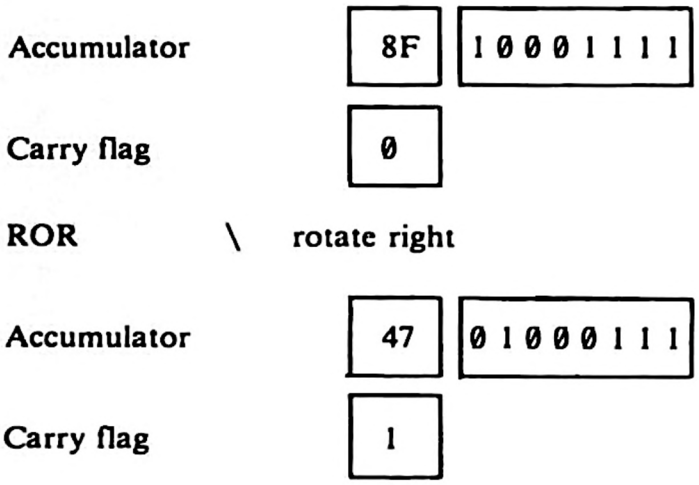
ROTATE RIGHT

Works just like ROL except the bits move to the right.


Byte
Before rotate:
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
C
Carry flag
After rotate:
C B_7 B_6 B_5 B_4 B_3 B_2 B_1
B_0

Example: ROR accumulator containing #8F.

```
CLC          \   clear Carry flag
LDA @#8F     \   load accumulator with #8F
```



LOGICALLY SPEAKING

If you need to shift (or rotate) the contents of a particular location *several times*, it is more efficient to load the value into the accumulator, shift (or rotate) that and store it back, than to manipulate the location directly.

For example, to rotate location #1234 to the right four times, we could use:

```
ROR #1234
ROR #1234
ROR #1234
ROR #1234
```

This uses twelve bytes of memory, four for the instructions and eight for addresses. Alternatively:

```
LDA #1234
ROR A
ROR A
ROR A
ROR A
STA #1234
```

uses two bytes less and is 25% quicker in operation.

So far we have only considered shifting and rotating single bytes. By using combinations of instructions it is possible to perform similar operations on two byte values such as #0123.

To perform an overall ASL on two bytes located at HIGH and LOW, ASL and ROL are used in conjunction:

```
ASL HIGH          \ shift bit 7 by LOW into Carry flag
ROL LOW           \ rotate it into bit 0 of HIGH
```

By exchanging the commands an overall LSR on the same two bytes can be performed:

```
LSR HIGH          \ shift bit 0 HIGH into Carry flag
ROR LOW           \ rotate it into bit 7 of LOW
```

Note that the bytes are manipulated in the reverse order because we wish to move the bits in the opposite direction. As with single byte shifts, the two byte values can be doubled or divided in half.

Two byte rotates to move the bits in a circular manner, are simply rotation operations performed twice! However, as with two byte shifts, it is important to get the byte rotation order correct.

A two byte ROR is performed with:

```
ROR HIGH          \ rotate bit 0 of HIGH into Carry flag
ROR LOW           \ and on into bit 7 of LOW
```

While two byte ROLs are implemented with:

```
ROL LOW           \ rotate bit 7 of LOW into Carry flag
ROL HIGH          \ and on into bit 0 of HIGH
```

Finally, moving back to single byte shifts, to shift the contents of the accumulator right one bit while preserving the sign bit, use the following technique:

```
TAY               \ save accumulator in Y register
ASL A             \ move sign bit (bit 7) into Carry flag
TYA               \ restore original value back into
                    accumulator
ROR A             \ rotate right moving sign bit back
                    into bit 7
```

The Y register has been used as a temporary store for the accumulator. We could have used the X register or a memory location with equal effect.

PRINTING BINARY!

Quite often, it is necessary to know the binary bit pattern that a register or memory location holds. This is particularly true in the case of the Status register when the condition of its flags can often provide a great deal of information about the way a program is running.

Program 24 shows how the binary value of a byte can be printed. It uses the Status register's contents at the time the program is RUN as an example.

Program 24

```
 10  REM * * SR IN BINARY * *
 15  REM * * HIMEM #9600 * *
 20  CODE = #9600
 30  FOR LOOP = 0 TO 20
 40    READ BYTE
 50    POKE CODE + LOOP, BYTE
 60  NEXT LOOP
 70
 80  REM * * M/C DATA * *
 90  DATA #08                :  REM PHP
100  DATA #68                :  REM PLA
110  DATA #8D, #00, #04      :  REM STA #400
120  DATA #A2, #08           :  REM LDX @#8
130  DATA #0E, #00, #04      :  REM ASL #400
140  DATA #A9, #00           :  REM LDA @#0
150  DATA #69, #00           :  REM ADC @#0
160  DATA #9D, #00, #04      :  REM STA #400, X
170  DATA #CA                :  REM DEX
180  DATA #D0, #F3           :  REM BNE -13
190  DATA #60                :  REM RTS
200
210  CLS
220  PRINT "Status Register"
230  PRINT "N V — B D I Z C"
240  CALL (CODE)
250  FOR LOOP = 8 TO 1 STEP -1
260    PRINT PEEK (#400 + LOOP);
270  NEXT LOOP
```

The Status register needs to be transferred into a memory location so that it can be manipulated. To do this, it must first be pushed onto the stack (line 90), then pulled into the accumulator (line 100) and *then* it can be stored in the user area (line 110). The X register is used to count the eight bits of the status byte, so it is initialized accordingly in line 120. The arithmetic shift left (line 130) moves the most significant bit of #400 into the Carry flag. The accumulator is loaded with #00 (line 140) and then the ADC instruction (line 150) is used to add 0 to it! No I haven't gone barmy, remember the Carry flag is taken into consideration during the ADC operation, therefore if it is set by the shifting procedure, the accumulator will now contain 1 (otherwise it will not be altered). Anyway, the result is saved in the user area (line 160) using indexed addressing. Because the X register was originally set to 8, and is also being used as the indexing register, the result of each shift operation is saved in reverse order. The loop is executed eight times while the X register is being decremented to zero. On returning to BASIC, the value of each flag is displayed by using a negative loop to extract the bit values from the user area.

To prove that the program really does work you might like to include a flag changing instruction inside the program. For example:

```
85  DATA #A9, #FF                  :  REM LDA @#FF set N clear Z
```

or:

```
85  DATA #A9, #00                  :  REM LDA @#00 clear N set Z
```

These will condition the flags as indicated in the REMs; remember to increase the 'DATA-loading' loop count by the extra number of bytes you insert (two in the above examples).

BIT

The instruction, BIT, allows individual bits of a specified memory location to be tested. It has an important feature in that it does *not* change the contents of either the accumulator or the memory location being tested, but, as you may have guessed, it conditions various flags within the Status register. Thus:

1. The Negative flag is loaded with the value of bit 7 of the location being tested.
2. The Overflow flag is loaded with the value of bit 6 of the location being tested.
3. The Zero flag is set if the AND operation between the accumulator and the memory location produces a zero.

By loading the accumulator with a mask it is possible to test any particular bit of a memory location. For example, to test location TEMP to see if bit 0 is clear the following could be used:

```
LDA @#1                         \ 00000001
BIT TEMP                        \ test bit 0
```

If bit 0 of TEMP contains a 0, the Zero flag will be set, otherwise it will remain clear, thus allowing BNE and BEQ to be used for testing purposes.

This masking procedure need only be used for testing bits 0 to 5 because bits 6 and 7 are automatically copied into the Negative and Overflow flags, which have their own test instructions.

```
BIT TEMP
BMI                             \ branch if bit 7 set
BPL                             \ branch if bit 7 clear
BVC                             \ branch if bit 6 set
BVS                             \ branch if bit 6 clear
```

15 Multiplication and Division

MULTIPLICATION

Performing multiplication in machine code is not too difficult provided that you have grasped what you have read so far. Unfortunately there are no multiplication instructions within the Oric's 6502 instruction set, therefore it is necessary to develop an algorithm to carry out this procedure.

Let's first look at the simplest method of multiplying two small values together. Consider the multiplication 5 × 6. We know the result is 30, but how did we obtain this? Simply by adding together six lots of five, in other words: 5 + 5 + 5 + 5 + 5 + 5 = 30. This is quite easy to implement:

Program 25

```
 10 REM * * SIMPLY MULTIPLY * *
 20 REM * * HIMEM #9600 * *
 30 CODE = #9600
 40 FOR LOOP = 0 TO 19
 50   READ BYTE
 60   POKE CODE + LOOP, BYTE
 70 NEXT LOOP
 80
 90 REM * * M/C DATA * *
100  DATA #A9, #00            : REM LDA @#0
110  DATA #8D, #00, #04       : REM STA #400
120  DATA #A2, #06            : REM LDX @#6
130  DATA #18                 : REM CLC
140  DATA #AD, #00, #04       : REM LDA #400
150  DATA #69, #05            : REM ADC @#5
160  DATA #8D, #00, #04       : REM STA #400
170  DATA #CA                 : REM DEX
180  DATA #D0, #F5            : REM BNE -11
190  DATA #60                 : REM RTS
200
```

```
210  REM * * SET UP & EXECUTE * *
220  CLS
230  CALL (CODE)
240  PRINT PEEK (#400)
```

All we have done here is to create a loop to add 5 to the contents of location #400 six times to produce the desired result! This method is reasonable for multiplying small values, but not particularly efficient for larger numbers.

At this point, it might be worth reviewing the usual procedure for multiplying two large decimal numbers together. Consider 123 × 150. We would approach this, (without calculators, please!) thus:

```
  123    (Multiplicand)
 ×150    (Multiplier)
 ----
  000    (Partial product 1)
 615     (Partial product 2)
123      (Partial product 3)
-----
18450    (Result or final product.)
```

The initial two values are termed the multiplicand and multiplier, and their product is formed by multiplying, in turn, each digit in the multiplier by the multiplicand. This results in a partial product, which is written such that its least significant digit sits directly below the multiplier digit to which it corresponds. When formation of all the partial products is completed, they are added together to give the final product or result.

We can apply this technique to binary numbers, starting off with two three bit values, 010 × 011

```
  010    (Multiplicand)
 ×011    (Multiplier)
 ----
  010    (Partial product 1)
 010     (Partial product 2)
000      (Partial product 3)
-----
00110    (Result)
```

Ignoring leading zeros, we obtain the result 110 (2 × 3 = 6). Moving on to our original decimal example, its binary equivalent is:

```
        01111011    123(#7B)
       ×10010110    150(#96)
        --------
        00000000
       01111011
      01111011
     00000000
    01111011
   000000000
  00000000
 01111011
 ---------------
 100100000010010    18450(#4812)
```

Hopefully you will have noticed that if the multiplier digit is a 0 it will result in the whole partial product being a line of zeros (anything multiplied by zero is zero). Therefore if a 0 is present in the multiplier it can simply be ignored *but* we must remember to shift the next partial product up past any 0s so that its least significant digit still corresponds to the correct 1 of the multiplier. This technique of shifting and ignoring can be used to write an efficient multiplication program.

Program 26

```
 10  REM * * SINGLE BYTE MULTIPLICATION * *
 20  REM * * GIVING A TWO BYTE ANSWER * *
 30  REM * * ENTER HIMEM #9600 FIRST * *
 40  REM * * BEFORE RUNNING THIS PROGRAM * *
 50  CODE = #9600
 60  REPEAT
 70    READ BYTE
 80    POKE CODE + OFFSET, BYTE
 90    OFFSET = OFFSET + 1
100  UNTIL BYTE = #60
110
120  REM * * M/C DATA * *
130
140  DATA #A2, #08                :  REM LDX @8
150  DATA #A9, #00                :  REM LDA @0
160  DATA #46, #71                :  REM LSR #71
170  DATA #90, #03                :  REM BCC +3
180  DATA #18                     :  REM CLC
190  DATA #65, #70                :  REM ADC #70
200  DATA #6A                     :  REM ROR A
210  DATA #66, #72                :  REM ROR #72
220  DATA #CA                     :  REM DEX
230  DATA #D0, #F3                :  REM BNE -12
240  DATA #85, #73                :  REM STA #73
250  DATA #60                     :  REM RTS
260
270  REM * * SET UP & RESULT * *
280  CLS
290  INPUT "MULTIPLICAND "; MCAND
300  POKE #70, MCAND
310  INPUT "MULTIPLIER "; MLIER
320  POKE #71, MLIER
330  CALL (CODE)
340  PRINT "RESULT IS :";
350  PRINT DEEK (#72)
```

This program takes two single byte numbers, multiplies them together storing the result (which may be 16 bits long) in zero page. Unlike the binary multiplication examples, it does not compute each partial product before adding them together, but totals the partial products as they are evaluated. This is a somewhat quicker method, because the final product is generated as soon as the last bit of the multiplier has been examined.

DIVISION

When performing the division of one number by another, we are actually calculating the number of times the second number can be subtracted from the first. Consider 125 ÷ 5:

```
                   25    (Quotient)
                 ____
(Divisor)      5 | 125    (Dividend)
                -1Ø
                ___
                 25
                 25
                 __
                  Ø    (Remainder)
```

Here, 5 can be subtracted from 1Ø twice, so we note the value 2 as part of the quotient. The 1Ø is brought down and subtracted from the first two digits of the dividend, leaving 2. Because 5 cannot be subtracted from 2 the remaining 5 of the dividend is brought down to give 25. 5 can be subtracted from this, without remainder, 5 times. Again this is recorded in the quotient, which now reflects the final result.

To divide binary numbers, this same procedure is pursued. The above example in binary would look like this:

```
        ØØØ11ØØ1
       _________
Ø1Ø1 | Ø11111Ø1
       Ø1Ø1
       ____
         1Ø
       ____
         1Ø1
         1Ø1
         ___
           Ø
         ___
           1Ø1
           1Ø1
           ___
             Ø
```

In fact, as you may see, dividing binary numbers is much simpler than dividing decimal numbers. If the divisor is less than or equal to the dividend the corresponding bit in the quotient will be a 1. If the subtraction is not possible a Ø is placed in the quotient, the next bit of the dividend is brought down, and the procedure repeated.

The following utility program divides two single byte values and indicates whether a remainder is present:

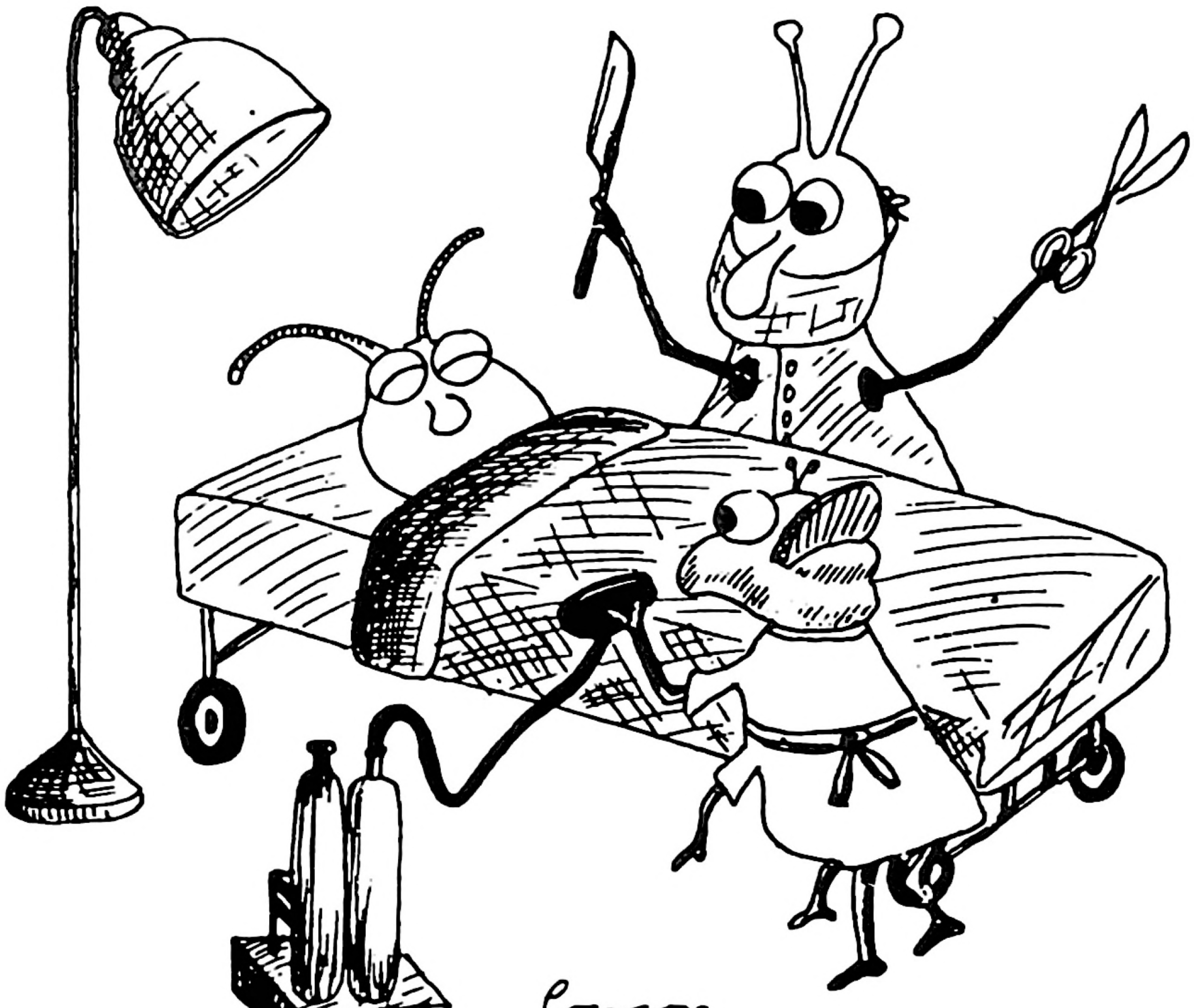
Program 27

```
 1Ø  REM * * SINGLE BYTE DIVIDE * *
 2Ø  REM * * GIVING A TWO BYTE ANSWER * *
 3Ø  REM * * ENTER HIMEM #96ØØ FIRST * *
 4Ø  REM * * BEFORE RUNNING THIS PROGRAM * *
 5Ø  CODE = #96ØØ
 6Ø  REPEAT
 7Ø    READ BYTE
 8Ø    POKE CODE + OFFSET, BYTE
 9Ø    OFFSET = OFFSET + 1
1ØØ  UNTIL BYTE = #6Ø
```

```
110
120  REM * * M/C DATA * *
130
140  DATA #A2, #08              :  REM LDX @8
150  DATA #A9, #00              :  REM LDA @0
160  DATA #06, #71              :  REM ASL #71
170  DATA #2A                   :  REM ROL A
180  DATA #C5, #70              :  REM CMP #70
190  DATA #90, #04              :  REM BCC +4
200  DATA #E5, #70              :  REM SBC #70
210  DATA #E6, #71              :  REM INC #71
220  DATA #CA                   :  REM DEX
230  DATA #D0, #F2              :  REM BNE −13
240  DATA #85, #72              :  REM STA #72
250  DATA #60                   :  REM RTS
260
270  REM * * SET UP & RESULT * *
280  CLS
290  INPUT "DIVIDEND "; DEND
300  POKE #70, DEND
310  INPUT "DIVISOR "; DSOR
320  POKE #71, DSOR
330  CALL (CODE)
340  PRINT "RESULT IS :";
350  PRINT PEEK (#71)
360  PRINT "REMAINDER :";
370  PRINT PEEK (#72)
```

This program uses the shift instructions of lines 160 and 170 as a two byte shift register in which the accumulator acts as the higher byte. The carry produced by ROL A is insignificant, in fact it is 0, and is eroded by the next ASL #71 procedure.

Appendices



I ASCII Codes

The 'American Standard Code for Information Interchange' (no wonder it's shortened to ASCII!), is used by virtually all microcomputers as a way of coding letters and a range of 'control characters' so that they may be handled by number conscious micros.

The ASCII codes 0–31 are known as the control codes, as they are used to control various aspects of the Oric's operation.

Table A1.1

Decimal	Hex	Meaning
0	0	NUL—Do nothing!
1	1	CTRL A
2	2	
3	3	Interrupt
4	4	Double height on/off
5	5	
6	6	Keyclick on/off
7	7	Bleep internal speaker
8	8	Move text cursor back a single space
9	9	Move text cursor forward a single space
10	A	Line feed
11	B	Vertical tab (move up a single line)
12	C	Clear text area
13	D	Carriage return (cursor to start of line)
14	E	Delete row
15	F	
16	10	Printer on/off
17	11	Cursor on/off
18	12	
19	13	Screen on/off
20	14	
21	15	
22	16	
23	17	
24	18	Cancel line
25	19	
26	1A	
27	1B	
28	1C	
29	1D	Inverse on/off
30	1E	
31	1F	

The numbers 32–127 are used to define letters, numbers and punctuation marks, and are detailed in Table A1.2.

Table A1.2

Decimal	Hex	ASCII	Decimal	Hex	ASCII
32	20	(space)	81	51	Q
33	21	!	82	52	R
34	22	"	83	53	S
35	23	#	84	54	T
36	24	$	85	55	U
37	25	%	86	56	V
38	26	&	87	57	W
39	27	'	88	58	X
40	28	(	89	59	Y
41	29	)	90	5A	Z
42	2A	*	91	5B	[
43	2B	+	92	5C	\
44	2C	,	93	5D	]
45	2D	–	94	5E	^
46	2E	.	95	5F	£
47	2F	/	96	60	©
48	30	0	97	61	a
49	31	1	98	62	b
50	32	2	99	63	c
51	33	3	100	64	d
52	34	4	101	65	e
53	35	5	102	66	f
54	36	6	103	67	g
55	37	7	104	68	h
56	38	8	105	69	i
57	39	9	106	6A	j
58	3A	:	107	6B	k
59	3B	;	108	6C	l
60	3C	<	109	6D	m
61	3D	=	110	6E	n
62	3E	>	111	6F	o
63	3F	?	112	70	p
64	40	@	113	71	q
65	41	A	114	72	r
66	42	B	115	73	s
67	43	C	116	74	t
68	44	D	117	75	u
69	45	E	118	76	v
70	46	F	119	77	w
71	47	G	120	78	x
72	48	H	121	79	y
73	49	I	122	7A	z
74	4A	J	123	7B	{
75	4B	K	124	7C	|
76	4C	L	125	7D	}
77	4D	M	126	7E	~
78	4E	N	127	7F	delete
79	4F	O	128	80	
80	50	P	129	81	

2 The 65Ø2

So far throughout this book we have been concerned with the software aspects of the Oric's 65Ø2, or in other words, how to program it! We could not really finish without having a glimpse at its hardware or physical features. For example, just how is it organized internally and how does it transfer data to and fro? While it is not absolutely vital to understand these features, an understanding of its design will enhance your new found knowledge.

Figure A2.1 shows a simplifed block diagram of the 65Ø2's design or *architecture* as it is more commonly called. If you study it many of the features will be readily recognizable. There are a few exceptions though, including three buses, the address bus, the data bus, and the control bus. You may well be wondering just what is meant by bus? It is not, as you may have thought, a number 19 bound for Highbury Barn—it's simply a collective term for a series of wires—or *tracks* as they are called on a *Printed Circuit Board* (PCB for short)—onto which a 1 or a Ø can be placed electronically.

By placing a series of 1s and Øs onto the eight lines of the data bus, a byte of information may be transferred to or from the address specified by the binary value present at that instant in time on the 16 lines of the address bus.

The control bus lines are responsible for carrying the numerous synchronization signals that are required for the Oric to operate.

EXECUTING INSTRUCTIONS

We can now examine just how the 65Ø2 fetches, interprets and executes each instruction. Firstly, the 65Ø2 must locate and read the next instruction of the machine code program. It does this by placing the current contents of the Program Counter onto the address bus and simultaneously placing a read signal on the appropriate control bus line. Almost instantaneously the instruction, or more correctly the byte that constitutes the instruction, is placed onto the data bus. The 65Ø2 then reads the contents of the data bus into a special internal, eight bit register, known as the Instruction Register (IR for short), which is used exclusively by the 65Ø2 to hold data waiting for processing. Once in the IR, the Control Unit interprets the instruction and then generates the various internal and external signals required to execute the instruction. For example, if the data byte fetched was #A5, the 65Ø2 would interpret this as LDA zero page, and would fetch the next byte of data and interpret this as the address at which the data to be placed into the accumulator is located. Each one of these operations would be performed in a manner similar to that already described.

Obviously instructions and data must be fetched in the correct sequence. To enable this to happen the Program Counter is provided with an automatic incrementing device. Each time the Program Counter's contents are placed onto the address bus the incrementer adds one to its contents, thus ensuring bytes are fetched and stored in the correct order.

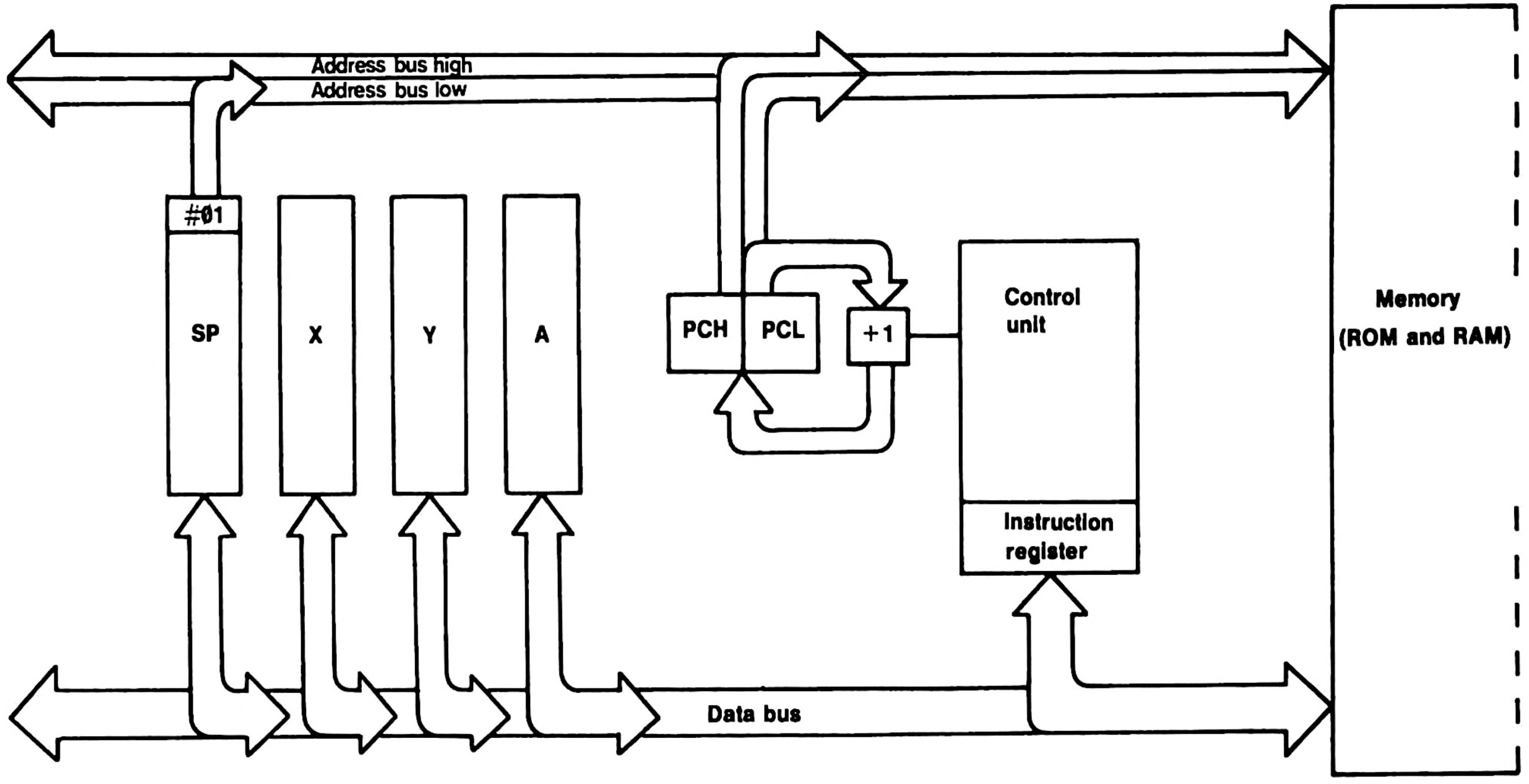


Figure A2.1 The 65Ø2—block diagram.

3 The Instruction Set

This section contains a full description of each of the 56 instructions that the 6502 is provided with. For ease of reference, the instructions are arranged in alphabetical order by mnemonic, and each description is broken down into the following six sections:

Introduction A brief one or two line description of the instruction's function.

Table This details the addressing modes available with the instruction, and lists the various opcodes, the total number of memory bytes required by each addressing mode, and finally the number of cycles that particular addressing mode takes to complete.

Status Shows the effect the execution of the instruction has on the Status register. The following codes are employed:

*	The flag is affected by the instruction but bits are undefined, being dependent on the byte's contents
1	The flag is set by the instruction
0	The flag is cleared by the instruction

If no code is indicated the flag remains unaltered by the instruction.

Operation A brief description of how the instruction operates together with details of its effect on the Status register.

Applications Some hints and tips on the sort of applications the instruction might be used for.

ADC

Add memory to accumulator with carry.

Addressing	Opcode	Bytes	Cycles
ADC @immediate	#69	2	2
ADC zero page	#65	2	3
ADC zero page, X	#75	2	4
ADC absolute	#6D	3	4
ADC absolute, X	#7D	4	4/5
ADC absolute, Y	#79	3	4/5
ADC (zero page, X)	#61	2	6
ADC (zero page), Y	#71	2	5/6

N	V	—	B	D	I	Z	C
*	*					*	*

Operation Adds the contents of the specified memory location to the current contents of the accumulator. If the Carry flag is set this is added to the result which is then stored in the accumulator. If the result is greater than #FF (255) the Carry flag is set. If the result is equal to zero the Zero flag is set. The contents of bit 7 of the accumulator are copied into the Status register. If overflow occurred from bit 6 to bit 7 the Overflow flag is set.

Applications Allows single, double and multibyte numbers to be added together. Overflow from one byte to another is provided by the Carry flag which is included in the addition.

AND

Logical AND of memory location with accumulator.

Addressing	Opcode	Bytes	Cycles
AND @immediate	#29	2	2
AND zero page	#25	2	3
AND zero page, X	#35	2	4
AND absolute	#2D	3	4
AND absolute, X	#3D	3	4/5
AND absolute, Y	#39	3	4/5
AND (zero page, X)	#21	2	6
AND (zero page), Y	#31	2	5

N	V	—	B	D	I	Z	C
*						*	

Operation Logically ANDs the corresponding bits of the accumulator with the specified value or contents of memory location. The result of the operation is stored in the accumulator but memory contents remain unaltered. If the result of the AND is Ø, the Zero flag is set. If the result leaves bit 7 set, the Negative flag is set. Otherwise both flags are cleared.

Applications Used to 'mask off' the unwanted bits of the accumulator.

```
AND @#FØ          \ masks off lower nibble, 1111ØØØØ
AND @#ØF          \ masks off higher nibble, ØØØØ1111
```

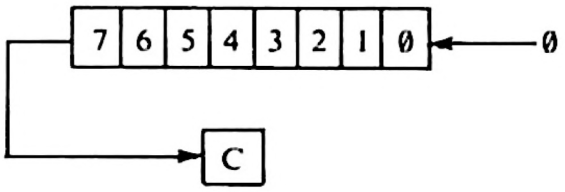
ASL

Shift contents of accumulator or memory left by one bit.

Addressing	Opcode	Bytes	Cycles
ASL accumulator	#0A	1	2
ASL zero page	#06	2	5
ASL zero page, X	#16	2	6
ASL absolute	#0E	3	6
ASL absolute, X	#1E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation Shuffles the bits in a specified location one bit left. Bit 7 moves into the carry, and a zero is placed into the vacated bit 0.



The Carry flag is set if bit 7 contained a 1 before the shift, and cleared if it contained 0. The Negative flag is set if bit 6 previously contained a 1. The Zero flag is set if the location holds #00 after the shift. (For this to occur it must previously have contained either #00 or #80).

Applications Multiplies the byte by two. Can be used to shift low nibble of byte into high nibble.

BCC

Branch if the Carry flag is clear (C = 0).

Addressing	Opcode	Bytes	Cycles
BCC relative	#90	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Carry flag is clear (C = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter. This gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Carry flag is set (C = 1) the branch does not occur and the next byte is ignored by the 6502.

Applications The Carry flag is conditioned by a number of instructions such as ADC, SBC, CMP, CPX and CPY, and a branch will occur if any of these result in clearing the flag. A 'forced' branch can be implemented using:

```
CLC                                  \ C = 0
BCC value                            \ 'jump'
```

BCS

Branch if the Carry flag is set (C = 1).

Addressing	Opcodes	Bytes	Cycles
BCS relative	#B0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Carry flag is set (C = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Carry flag is clear (C = 0) the branch does not occur and the next byte is ignored by the 6502.

Applications As with BCC but the branch will only take place if an operation results in the Carry flag being set. A 'forced' branch can be implemented with:

```
SEC                                  \ C = 1
BCS set                              \ 'jump'
```

BEQ

Branch if the Zero flag is set (Z = 1).

Addressing	Opcode	Bytes	Cycles
BEQ relative	#F0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Zero flag is set (Z = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new program address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Zero flag is clear (Z = 0) the branch does not occur and the next byte is ignored by the 6502.

Applications Used to cause a branch when the Zero flag is set. This happens when an operation results in zero (e.g. LDA @0). The BEQ command is used frequently after a comparison instruction, for example:

```
CMP @"?"
BEQ Questionmark
```

If the comparison succeeds the Zero flag is set therefore BEQ will work.

BIT

Test memory bits.

Addressing	Opcode	Bytes	Cycles
BIT zero page	#24	2	3
BIT absolute	#2C	3	4

N	V	—	B	D	I	Z	C
*	*					*	

Operation The BIT operation affects only the Status register, the accumulator and the specified memory location are unaltered. Bit 7 and bit 6 of the memory byte are copied directly into N and V respectively. The Zero flag is conditioned after a logical bitwise AND between the accumulator and the memory byte. If accumulator AND memory results in zero then Z = 1, otherwise Z = 0.

Applications Often used in conjunction with BPL/BMI or BVS/BVC to test bits 7 and 6 of a memory location and to cause a branch depending on their condition.

BMI

Branch if the Negative flag is set (N = 1).

Addressing	Opcode	Bytes	Cycles
BMI relative	#30	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Negative flag is set (N = 1) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Negative flag is clear (N = 0) the branch does not occur and the next byte is ignored by the 6502.

Applications Generally after an operation has been performed (i.e. LDA, LDX etc.) the most significant bit of the register is copied into the Negative flag position. If it is set then a branch will occur using BMI. The 'minus' part of the mnemonic denotes this instruction's importance when using signed arithmetic—where bit 7 is used to denote the sign of a number in two's complement form.

BNE

Branch if the Zero flag is clear (Z = 0).

Addressing	Opcode	Bytes	Cycles
BNE relative	#D0	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Zero flag is clear (Z = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Zero flag is set (Z = 1) the branch does not occur and the next byte is ignored by the 6502.

Applications Used to cause a branch when the Zero flag is clear. It's often used, in conjunction with a decrementing counter, as a loop controlling command.

```
DEX
BNE AGAIN
```

will continue branching back to AGAIN until X = 0 and the Zero flag is set.

BPL

Branch if the Negative flag is clear (N = 0).

Addressing	Opcode	Bytes	Cycles
BPL relative	#10	2	3/4/5

N	V	—	B	D	I	Z	C

Operation If the Negative flag is clear (N = 0) the byte following the instruction is interpreted as a two's complement number and is added to the current contents of the Program Counter; this gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Negative flag is set (N = 1) the branch does not occur and the next byte is ignored by the 6502.

Applications Generally after an operation has been performed (i.e. LDA, ROL, CPX etc.) the most significant bit of the register is copied into the Negative flag position. If it is clear then a branch will occur if BPL is used. The 'plus' part of the mnemonic denotes the instruction's importance when using signed arithmetic, where bit 7 is used to indicate the sign of a number in two's complement form. If a decrementing counter is being used in a loop this branch instruction allows the loop to execute when the counter reaches zero.

```
DEX
BPL again
```

This loop will finish when X is decremented from 0 to #FF because #FF = 1111 1111 binary, where bit 7 is set.

BRK

Software forced BREAK.

Addressing	Opcode	Bytes	Cycles
BRK implied	#00	1	7

N	V	—	B	D	I	Z	C
			1				

Operation The Program Counter address plus one is pushed onto the stack, followed by the contents of the Status register.

Applications Used as a software interrupt. This instruction should be avoided on the Oric as it will result in a 'hang-up'!

BVC

Branch if the Overflow flag is clear (V = Ø).

Addressing	Opcode	Bytes	Cycles
BVC relative	#5Ø	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Overflow flag is clear (V = Ø) the byte following the instruction is interpreted as a two's complement number and added to the current contents of the Program Counter. This gives the new address from which the program will now execute. This allows a branch of either 126 bytes back or 129 bytes forward. If the Overflow flag is set (V = 1) the branch does not take place and the next byte is ignored by the 65Ø2.

Applications Used to detect an overflow from bit 6 into bit 7 (i.e. a carry from bit 6 to bit 7) when using signed arithmetic. When using signed arithmetic two numbers of opposite sign *cannot* overflow, however numbers of the same sign *can* overflow. For example:

```
     Ø1ØØ1111          (#4F)
  +  Ø1ØØØØØØ          (#4Ø)
     ────────
     1ØØØ1111          (−#71)
     ↑
     └─Overflow from bit 6 to bit 7
```

The result is now negative which is, of course, absurd! Similarly adding two large negative numbers can produce a positive result. In fact overflow can occur in the following situations:

1. Adding large positive numbers.
2. Adding large negative numbers.
3. Subtracting a large negative number from a large positive number.
4. Subtracting a large positive number from a large negative number.

The Overflow flag is used to signal this overflow from bit 6 to bit 7 and therefore, in signed arithmetic, a change in sign. If it is clear no overflow has occurred and BVC will cause a branch.

A 'forced' branch may be implemented using:

```
CLV                        \ clear V
BVC Forced                 \ 'jump'
```

BVS

Branch if the Overflow flag is set (V = 1).

Addressing	Opcode	Bytes	Cycles
BVS relative	#70	2	2/3/4

N	V	—	B	D	I	Z	C

Operation If the Overflow flag is set (V = 1) the byte following the instruction is interpreted as a two's complement number and added to the current contents of the Program Counter. This gives the new address from which the program will now execute, allowing a branch of either 126 bytes back or 129 bytes forward. If the Overflow flag is clear (V = 0) the branch does not occur and the next byte is ignored by the 6502.

Applications Used to cause a branch if the sign of a number has been changed. In most instances this will only matter if signed arithmetic is being employed. See BVC for more details.

CLC

Clear the Carry flag (C = 0).

Addressing	Opcode	Bytes	Cycles
CLC implied	#18	1	2

N	V	—	B	D	I	Z	C
							0

Operation The Carry flag is cleared by setting it to zero.

Applications Should always be used before adding two numbers together as the Carry flag's contents are taken into account by ADC. A 'forced' branch may be implemented with:

```
CLC                    \ Clear C
BCC clear              \ and 'jump'
```

CLD

Clear the Decimal flag (D = Ø).

Addressing	Opcode	Bytes	Cycles
CLD implied	#D8	1	2

N	V	—	B	D	I	Z	C
				Ø			

Operation The Decimal flag is cleared by setting it to zero.

Applications Used to make 65Ø2 work in normal hexadecimal mode as opposed to decimal mode.

CLI

Clear the Interrupt flag (I = Ø).

Addressing	Opcode	Bytes	Cycles
CLI implied	#58	1	2

N	V	—	B	D	I	Z	C
					Ø		

Operation The Interrupt flag is cleared by setting it to zero.

Applications Causes any interrupts on the IRQ line to be processed immediately after completion of current instruction.

CLV

Clear the Overflow flag (V = Ø).

Addressing	Opcode	Bytes	Cycles
CLV implied	#B8	1	2

N	V	—	B	D	I	Z	C
	Ø						

Operation The Overflow flag is cleared by setting it to zero.

Applications Used to clear the Overflow flag after an overflow from bit 6 to bit 7. In most instances this is only important if signed arithmetic is being used.

CMP

Compare contents of memory with contents of the accumulator.

Addressing	Opcode	Bytes	Cycles
CMP @immediate	#C9	2	2
CMP zero page	#C5	2	3
CMP zero page, X	#D5	2	4
CMP absolute	#CD	3	4
CMP absolute, X	#DD	3	4/5
CMP absolute, Y	#D9	3	4/5
CMP (zero page, X)	#C1	2	6
CMP (zero page), Y	#D1	2	5/6

N	V	—	B	D	I	Z	C
*						*	*

Operation The contents of the specified memory location (or immediate value) are subtracted from the contents of the accumulator. The contents of the memory location and accumulator are *NOT* altered, but the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction, the 65Ø2 first sets the Carry flag and then adds the two's complement value of the memory location's contents to the accumulator's contents. If both values are equal (memory = accumulator) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the accumulator (memory < accumulator) the Zero flag is cleared and the Carry flag set. If memory contents are greater than the accumulator (memory > accumulator) then both the Zero flag and Carry flag are cleared. If unsigned binary is being used the Negative flag is also set. If signed binary is being used the Overflow flag should be checked in conjunction with the Negative flag to test for a 'true' negative result.

Applications Should be used to test for intermediate values that cannot be tested directly from the Status register. For example:

```
CMP @#ØØ
BEQ AWAY
```

is a waste of two bytes, as the Zero flag will be set if the accumulator contains #ØØ, therefore all that is needed is : BEQ AWAY.

CPX

Compare contents of memory with contents of the X register.

Addressing	Opcode	Bytes	Cycles
CPX @immediate	#E0	2	2
CPX zero page	#E4	2	3
CPX absolute	#EC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Operation The contents of the specified memory location (or immediate value) are subtracted from the contents of the X register. The contents of the memory location and X register are *NOT* altered, instead the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction the 6502 first sets the Carry flag and then adds the two's complement value of the memory location to the contents of the X register. If both values are equal (memory = X register) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the X register (memory < X register) the Zero flag is cleared but the Carry flag remains set. If memory contents are greater than the X register (memory > X register) then both Zero and Carry flags are cleared. If unsigned binary is being used then the Negative flag is set.

Applications Should be used to test for intermediate values which cannot be tested directly from the Status register. For example, to test the X register's contents during use as a loop counter try:

```
        LDX @#E0        \ load X with #E0
AGAIN   DEX             \ decrement X
        CPX @#87        \ has X reached #87?
        BNE AGAIN       \ no, go to AGAIN
```

CPY

Compare contents of memory with contents of the Y register.

Addressing	Opcode	Bytes	Cycles
CPY @immediate	#CØ	2	2
CPY zero page	#C4	2	3
CPY absolute	#CC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Operation The contents of the specified memory location (or immediate value) are subtracted from the contents of the Y register. The contents of the memory location and Y register are *NOT* altered, instead the Negative, Zero and Carry flags are conditioned according to the result of the subtraction. To perform this subtraction the 65Ø2 first sets the Carry flag and then adds the two's complement value of the memory location to the contents of the Y register. If both values are equal (memory = Y register) the Zero flag is set and the Carry flag remains set. If the contents of memory are less than the Y register (memory < Y register) the Zero flag is cleared but the Carry flag remains set. If memory contents are greater than the Y register (memory > Y register) then both Zero and Carry flags are cleared. If unsigned binary is being used then the Negative flag is set.

Applications Should be used to test for intermediate values which cannot be tested directly from the Status register. For example, to test the Y register's contents during use as a loop counter try:

```
        LDY @#EØ        \ load Y with #EØ
AGAIN   DEY             \ decrement Y
        CPY @#87        \ has Y reached #87?
        BNE AGAIN       \ no, go to AGAIN
```

DEC

Decrement memory contents by one.

Addressing	Opcode	Bytes	Cycles
DEC zero page	#C6	2	5
DEC zero page, X	#D6	2	6
DEC absolute	#CE	3	6
DEC absolute, X	#DE	3	7

N	V	—	B	D	I	Z	C
*						*	

Operation The byte at the address specified is decremented by one (MEMORY = MEMORY −1). If the result of the operation is zero the Zero flag will be set. Bit 7 of the byte is copied into the Negative flag.

Applications Used to subtract one from a counter stored in memory.

DEX

Decrement contents of X register by one.

Addressing	Opcode	Bytes	Cycles
DEX implied	#CA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation One is subtracted from the value currently held in the X register (X = X − 1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag (N = 0 if X < #80 ; N = 1 if X > #7F). The Carry flag is not affected by the instruction.

Applications Used with indexed addressing when the X register acts as an offset from a base address, allowing a sequential set of bytes to be accessed. Invariably used to decrement the X register when being used as a loop counter, branching until X = 0 (Z = 1).

DEY

Decrement contents of Y register by one.

Addressing	Opcode	Bytes	Cycles
DEY implied	#88	1	2

N	V	—	B	D	Z	C
*					*	

Operation One is subtracted from the value currently held in the Y register (Y = Y − 1). If the result of the operation is zero the Zero flag is set. Bit 7 is copied into the Negative flag (N = 0 if Y < #80;N = 1 if Y > #7F). The Carry flag is not affected by the instruction.

Applications Used with indexed addressing when the Y register acts as an offset from a base address allowing a sequential set of bytes to be accessed. Invariably used to decrement the Y register when being used as a loop counter, branching until Y = 0 (Z = 1).

EOR

Accumulator exclusively ORed with memory.

Addressing	Opcode	Bytes	Cycles
EOR @immediate	#49	2	2
EOR zero page	#45	2	3
EOR zero page, X	#55	2	4/5
EOR absolute	#4D	3	4
EOR absolute, X	#5D	3	4/5
EOR absolute, Y	#59	3	4/5
EOR (zero page, X)	#41	2	6
EOR (zero page), Y	#51	2	5/6

N	V	—	B	D	Z	C
*					*	

Operation Performs a bitwise exclusive OR between the corresponding bits in the accumulator and the specified memory byte. If the result, which is stored in the accumulator, is zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Used to complement or invert a data byte.

INC

Increment memory contents by one.

Addressing	Opcode	Bytes	Cycles
INC zero page	#E6	2	5
INC zero page, X	#F6	2	6
INC absolute	#EE	3	6
INC absolute, X	#FE	3	7

N	V	—	B	D	I	Z	C
*						*	

Operation The byte at the address specified is incremented by one. If the address holds zero after the operation the Zero flag is set. Bit 7 of the byte is copied into the Negative flag.

Applications Add one to a counter stored in memory.

INX

Increment contents of X register by one.

Addressing	Opcode	Bytes	Cycles
INX implied	#E8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation One is added to the value currently in the X register (X = X + 1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag (N = 0 if X < #80 ; N = 1 if X > #7F). The Carry flag is not affected by the instruction.

Applications Used with indexed addressing when the X register acts as an offset from a base address, and allows a sequential set of bytes to be accessed. Often used as a counter to control the number of times a loop of instructions is executed.

INY

Increment contents of Y register by one.

Addressing	Opcode	Bytes	Cycles
INY implied	#C8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation One is added to the value currently held in the Y register (Y = Y + 1). If the result of the operation is zero the Zero flag will be set. Bit 7 is copied into the Negative flag. The Carry flag is not affected.

Applications Used with indexed addressing when the Y register acts as an offset from a base address, allowing a sequential set of bytes to be accessed. Often used as a counter to control the number of times a loop is executed.

JMP

Jump to a new location.

Addressing	Opcode	Bytes	Cycles
JMP absolute	#4C	3	3
JMP (indirect)	#6C	3	3

N V — B D I Z C

Operation In an absolute JMP the two bytes following the instruction are placed into the Program Counter. In an indirect jump the two bytes located at the two byte address following the instruction are loaded into the Program Counter.

Applications Transfers control, unconditionally, to another part of a program stored anywhere in memory.

JSR

Jump, save return address.

Addressing	Opcode	Bytes	Cycles
JSR absolute	#20	3	6

N V — B D I Z C

Operation Acts as a subroutine call, transferring program control to another part of memory until an RTS is encountered. The current contents of the Program Counter plus two are pushed onto the stack. The Stack Pointer is incremented twice. The absolute address following the instruction is placed into the Program Counter and program execution continues from this new address.

Applications Allows large repetitive sections of programs to be entered once, out of the way of the main program, and called as subroutines as often as required.

LDA

Load the accumulator with the specified byte.

Addressing	Opcode	Bytes	Cycles
LDA @immediate	#A9	2	2
LDA zero page	#A5	2	3
LDA zero page, X	#B5	2	4
LDA absolute	#AD	3	4
LDA absolute, X	#BD	3	4/5
LDA absolute, Y	#B9	3	4/5
LDA (zero page, X)	#A1	2	6
LDA (zero page), Y	#B1	2	5/6

N	V	—	B	D	I	Z	C
*						*	

Operation Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the accumulator. If the value loaded is zero then the Zero flag is set. Bit 7 is copied into the Negative flag position.

Applications Probably the most frequently used instruction, it allows for general data movement and facilitates all logical and arithmetic operations.

LDX

Load the X register with the specified byte.

Addressing	Opcode	Bytes	Cycles
LDX @immediate	#A2	2	2
LDX zero page	#A6	2	3
LDX zero page, Y	#B6	2	4
LDX absolute	#AE	3	4
LDX absolute, Y	#BE	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Operation Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the X register. If the value loaded is zero then the Zero flag is set. Bit 7 is copied into the Negative flag position.

Applications General transfer of data for processing or storage. Also allows a loop counter to be set to its start value.

LDY

Load the Y register with the specified byte.

Addressing	Opcode	Bytes	Cycles
LDY @immediate	#AØ	2	2
LDY zero page	#A4	2	3
LDY zero page, X	#B4	2	4
LDY absolute	#AC	3	4
LDY absolute, X	#BC	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Operation Places the value immediately following the instruction, or the contents of the location specified after the instruction, into the Y register. If the value loaded is zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications General transfer of data for processing or storage. Also allows a loop counter to be set to its start value.

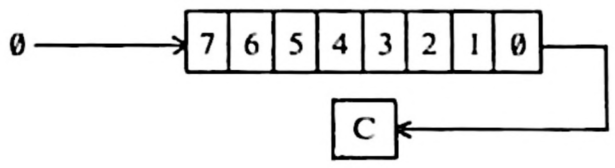
LSR

Logically shift the specified byte right one bit.

Addressing	Opcode	Bytes	Cycles
LSR accumulator	#4A	1	2
LSR zero page	#46	2	5
LSR zero page, X	#56	2	6
LSR absolute	#4E	3	6
LSR absolute, X	#5E	3	7

N	V	—	B	D	I	Z	C
0						*	*

Operation Moves the contents of the specified byte right by one position, putting a 0 in bit 7 and bit 0 into the Carry flag.



The Negative flag is cleared, and the Carry flag is conditioned by the contents of bit 0. The Zero flag is set if the specified byte now holds zero (in which case it must previously have contained #00 or #01).

Applications Divides a byte value by two (if D = 0) with its remainder shifting into the Carry flag position. Can also be used to shift the high nibble of a byte into the low nibble.

NOP

No operation.

Addressing	Opcode	Bytes	Cycles
NOP implied	#EA	1	2

N	V	—	B	D	I	Z	C

Operation Does nothing except increment the Program Counter.

Applications Provides a two cycle delay.

ORA

Logical OR of a specified byte with the accumulator.

Addressing	Opcode	Bytes	Cycles
ORA @immediate	#09	2	2
ORA zero page	#05	2	3
ORA zero page, X	#15	2	4
ORA absolute	#0D	3	4
ORA absolute, X	#1D	3	4/5
ORA absolute, Y	#19	3	4/5
ORA (zero page, X)	#01	2	6
ORA (zero page), Y	#11	2	5

N	V	—	B	D	I	Z	C
*						*	

Operation Logically ORs the corresponding bits of the accumulator with the specified value, or contents of a memory location. The result of the operation is stored in the accumulator. If the result leaves bit 7 set the Negative flag is set, otherwise it is cleared.

Applications Used to 'force' certain bits to contain a one. For example:

```
    ORA @#80                              \ 10000000 binary
```

will ensure bit 7 is set.

PHA

Push the accumulator contents onto the 'top' of the stack.

Addressing	Opcode	Bytes	Cycles
PHA implied	#48	1	3

N	V	—	B	D	I	Z	C

Operation The contents of the accumulator are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications Allows bytes of memory to be saved temporarily. The index registers can be saved by first transferring them to the accumulator; memory bytes are saved by first loading them into the accumulator. Bytes are recovered with PLA.

PHP

Push the Status register's contents onto the top of the stack.

Addressing	Opcode	Bytes	Cycles
PHP implied	#Ø8	1	3

N	V	—	B	D	I	Z	C

Operations The contents of the Status register are copied into the position indicated by the Stack Pointer. The Stack Pointer is then decremented by one.

Applications Allows the conditions of the flags to be saved, perhaps prior to a subroutine call, so that the same conditions can be restored with PLP on return.

PLA

Pull the 'top' of the stack into the accumulator.

Addressing	Opcode	Bytes	Cycles
PLA implied	#68	1	4

N	V	—	B	D	I	Z	C
*						*	

Operation The Stack Pointer is incremented by one, and the byte contained at this position in the stack is copied into the accumulator. If the byte is #ØØ the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Complements the operation of PHA to retrieve data previously pushed onto the stack.

PLP

Pull the 'top' of the stack into the Status register.

Addressing	Opcode	Bytes	Cycles
PLP implied	#28	1	4

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Operation The Stack Pointer is incremented by one and the byte contained at this position is copied into the Status register.

Applications Complements the operation of PHP to retrieve the previously pushed contents of the Status register, or to condition certain flags from a defined byte previously pushed onto the stack via the accumulator.

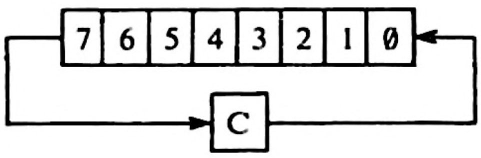
ROL

Rotate either the accumulator or a memory byte left by one bit with the Carry flag.

Addressing	Opcode	Bytes	Cycles
ROL accumulator	#2A	1	2
ROL zero page	#26	2	5
ROL zero page, X	#36	2	6
ROL absolute	#2E	3	6
ROL absolute, X	#3E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation The specified byte and the contents of the Carry flag are rotated left by one bit in a circular manner.



Bit 7 is rotated into the Carry flag, with the flag's previous contents moving into bit Ø. The remaining bits are shuffled left. The Negative flag is set if bit 6 previously held 1; cleared otherwise. The Carry flag is conditioned by bit 7, and if the specified byte now holds zero the Zero flag is set.

Applications Used in conjunction with ASL, ROL can be used to double the value of multibyte numbers, as the Carry bit is used to propagate the overflow from one byte to another. It may also be used before testing the Negative, Zero and Carry flags to determine the state of specific bits.

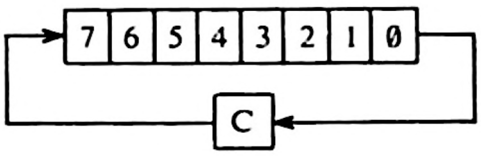
ROR

Rotate either the accumulator or a memory byte right by one bit with the Carry flag.

Addressing	Opcode	Bytes	Cycles
ROR accumulator	#6A	1	2
ROR zero page	#66	2	5
ROR zero page, X	#76	2	6
ROR absolute	#6E	3	6
ROR absolute, X	#7E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Operation The specified byte and the contents of the Carry flag are rotated right by one bit in a circular manner.



Bit 0 is rotated into the Carry flag with the flag's previous contents moving into the bit 7 position. The remaining bits are shuffled right. The Negative flag is set if the Carry flag was set previously; otherwise it is cleared. If bit 0 contained a 1 the Carry flag will now also be set. If the specified byte now holds zero the Zero flag is set.

Applications Used in conjunction with LSR, ROR can be used to halve the value of multibyte numbers. It may also be used before testing the Negative, Zero and Carry flags to determine the contents of specific bits.

RTI

Return from interrupt.

Addressing	Opcode	Bytes	Cycles
RTI implied	#40	1	6

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Operation This instruction expects to find three bytes on the stack. The first byte is pulled from the stack and placed into the Status register—thus conditioning all flags. The next two bytes are placed into the Program Counter. The Stack Pointer is incremented as each byte is pulled.

Applications Used to restore control to a program after an interrupt has occurred. On detecting the interrupt, the processor will have pushed the Program Counter and Status register onto the stack.

RTS

Return from subroutine.

Addressing	Opcode	Bytes	Cycles
RTS implied	#60	1	6

N	V	—	B	D	I	Z	C

Operation The two bytes on the top of the stack are pulled, incremented by one, and placed into the Program Counter. Program execution continues from this address. The Stack Pointer is incremented by two.

Applications Returns control from a subroutine to the calling program. It should therefore be the last instruction of a subroutine.

SBC

Subtract specified byte from the accumulator with borrow.

Addressing	Opcode	Bytes	Cycles
SBC @immediate	#E9	2	2
SBC zero page	#E5	2	3
SBC zero page, X	#F5	2	4
SBC absolute	#ED	3	4
SBC absolute, X	#FD	3	4/5
SBC absolute, Y	#F9	3	4/5
SBC (zero page, X)	#E1	2	6
SBC (zero page), Y	#F1	2	5/6

N	V	—	B	D	I	Z	C
*	*					*	*

Operation Subtracts the immediate value, or the byte contained at the specified address, from the contents of the accumulator. If the value is greater than the contents of the accumulator it will 'borrow' from the Carry flag, which should be set at the onset (only) of a subtraction. If the Carry flag is clear after the subtraction, a borrow has occurred. If the result is #00 the Zero flag is set. The contents of bit 7 are copied into the accumulator and V is set if an overflow from bit 6 to bit 7 occurred.

Applications Allows single, double and multibyte numbers to be subtracted from one another.

SEC

Set the Carry flag (C = 1).

Addressing	Opcode	Bytes	Cycles
SEC implied	#38	1	2

N	V	—	B	D	I	Z	C
							1

Operation A one is placed into the Carry flag bit position.

Applications Should always be used at the onset of subtraction as the Carry flag is taken into account by SBC.

SED

Set the Decimal mode flag (D = 1).

Addressing	Opcode	Bytes	Cycles
SED implied	#F8	1	2

N	V	—	B	D	I	Z	C
				1			

Operation A one is placed into the Decimal flag position.

Applications Puts the Beeb in decimal mode, in which Binary Coded Decimal (BCD) arithmetic is performed. The Carry flag now denotes a carry of hundreds, as the maximum value that can be encoded in a single BCD byte is 99.

SEI

Set the Interrupt disable flag (I = 1).

Addressing	Opcode	Bytes	Cycles
SEI implied	#78	1	2

N	V	—	B	D	I	Z	C
					1		

Operation A one is placed into the Interrupt flag position.

Applications When this flag is set *no* interrupts occurring on the IRQ line are processed. However NMI interrupts are processed, as are BREAKS.

STA

Store the accumulator's contents in a memory location.

Addressing	Opcode	Bytes	Cycles
STA zero page	#85	2	3
STA zero page, X	#95	2	4
STA absolute	#8D	3	4
STA absolute, X	#9D	3	5
STA absolute, Y	#99	3	5
STA (zero page, X)	#81	2	6
STA (zero page), Y	#91	2	6

N	V	—	B	D	I	Z	C

Operations The contents of the accumulator are copied into the specified memory location.

Applications To save the contents of the accumulator, or to initialize areas of memory to specific values. Used in conjunction with LDA, blocks of data can be transferred from one area of memory to another.

STX

Store the X register's contents in memory.

Addressing	Opcode	Bytes	Cycles
STX zero page	#86	2	3
STX zero page, X	#96	2	4
STX absolute	#8E	3	4

N	V	—	B	D	I	Z	C

Operation The contents of the X register are copied into the specified memory location.

Applications To save the X register's contents, or to initialize areas of memory to specific values.

STY

Store the Y register's contents in memory.

Addressing	Opcode	Bytes	Cycles
STY zero page	#84	2	3
STY zero page, X	#94	2	4
STY absolute	#8C	3	4

N	V	—	B	D	I	Z	C

Operations The contents of the Y register are copied into the specified memory location.

Applications To save the Y register's contents, or to initialize areas of memory to specific values.

TAX

Transfer the accumulator's contents into the X register.

Addressing	Opcode	Bytes	Cycles
TAX implied	#AA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation The contents of the accumulator are copied into the X register. If the X register now holds zero, the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Allows the accumulator's values to be saved temporarily, or perhaps used to seed the X register as a loop counter. Often used after PLA to restore the X register's contents previously pushed onto the stack.

TAY

Transfer accumulator's contents into the Y register.

Addressing	Opcode	Bytes	Cycles
TAY implied	#A8	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation The contents of the accumulator are copied into the Y register. If the Y register now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Allows the accumulator's values to be saved temporarily, or perhaps used to seed the Y register as a loop counter. Often used after PLA to restore the Y register's contents previously pushed onto the stack.

TSX

Transfer the Stack Pointer's contents into the X register.

Addressing	Opcode	Bytes	Cycles
TSX implied	#BA	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation The contents of the Stack Pointer are copied into the X register. If X now holds zero, the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications To calculate the amount of space left on the stack, or to save its current position while the stack contents are checked.

TXA

Transfer the X Register's contents into the accumulator.

Addressing	Opcode	Bytes	Cycles
TXA implied	#8A	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation The contents of the X register are copied into the accumulator. If the accumulator now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Allows the X register's contents to be manipulated by logical or arithmetic instructions. Followed by a PHA it allows the X register's value to be saved on the stack.

TXS

Transfer the X Register's contents into the Stack Pointer.

Addressing	Opcode	Bytes	Cycles
TXS implied	#9A	1	2

N	V	—	B	D	I	Z	C

Operation The contents of the X register are copied into the Stack Pointer.

Applications Allows the contents of the Stack Pointer to be set or reset to a specific value. For example:

```
LDX @#FF
TXS
```

will 'clear' the stack and reset the Stack Pointer.

TYA

Transfer the Y register's contents into the accumulator.

Addresssing	Opcode	Bytes	Cycles
TYA implied	#98	1	2

N	V	—	B	D	I	Z	C
*						*	

Operation The contents of the Y register are copied into the accumulator. If the accumulator now holds zero the Zero flag is set. Bit 7 is copied into the Negative flag.

Applications Allows the Y register's contents to be manipulated by logical or arithmetic instructions. When followed by a PHA, it allows the Y register's value to be saved on the stack.

4 Branch Calculators

The branch calculators are used to give branch values in hex. First, count the number of bytes you need to branch. Then locate this number in the centre of the appropriate table, and finally, read off the high and low hex nibbles from the side column and top row respectively.

Example For a backward branch of 16 bytes:

Locate 16 in the centre of Table A4.1 (bottom row), then read off high nibble (#F) and low nibble (#0) to give displacement value (#F0).

Table A4.1 Backward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Table A4.2 Forward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

5 6502 Opcodes

All numbers are hexadecimal.

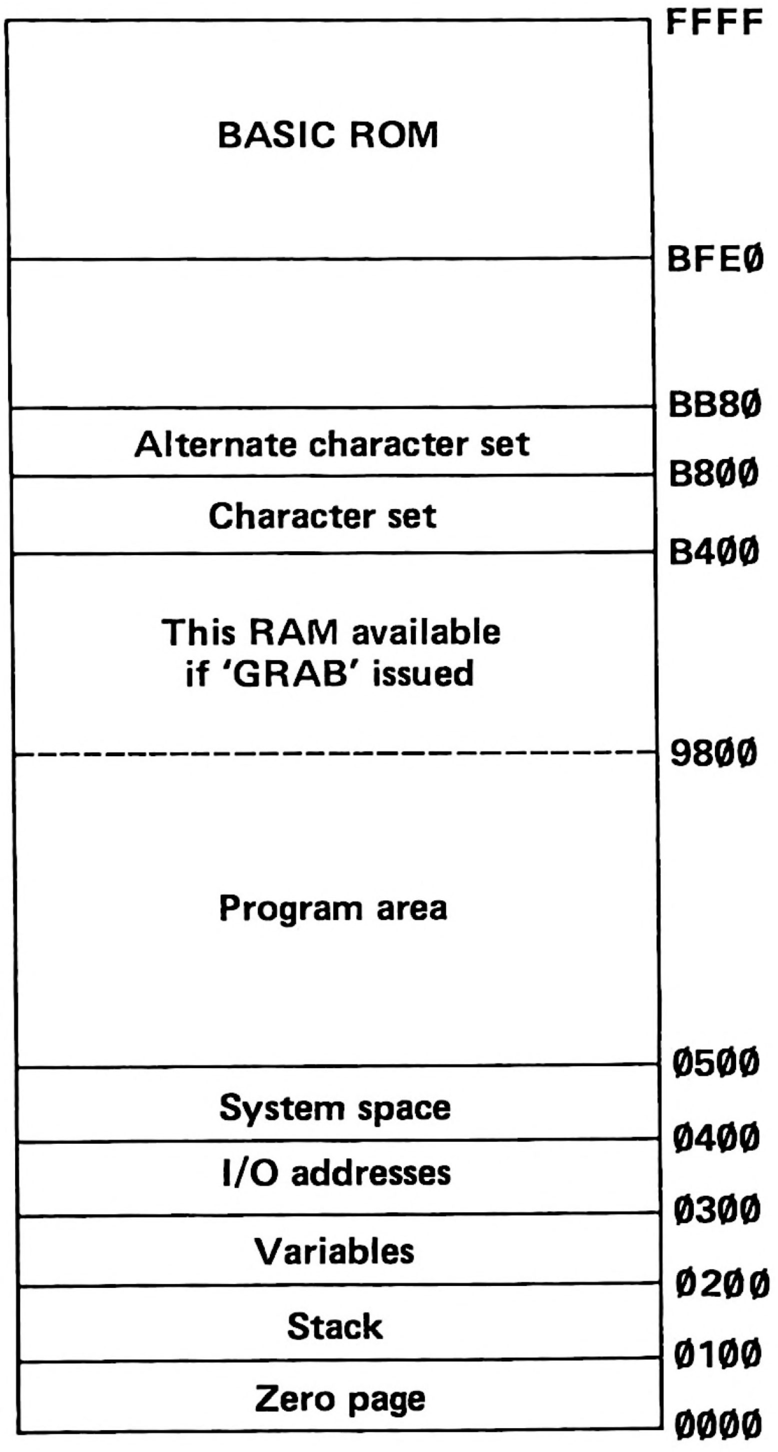
00	BRK implied
01	ORA (zero page, X)
02	Future expansion
03	Future expansion
04	Future expansion
05	ORA zero page
06	ASL zero page
07	Future expansion
08	PHP implied
09	ORA #immediate
0A	ASL accumulator
0B	Future expansion
0C	Future expansion
0D	ORA absolute
0E	ASL absolute
0F	Future expansion
10	BPL relative
11	ORA (zero page), Y
12	Future expansion
13	Future expansion
14	Future expansion
15	ORA zero page, X
16	ASL zero page, X
17	Future expansion
18	CLC implied
19	ORA absolute, Y
1A	Future expansion
1B	Future expansion
1C	Future expansion
1D	ORA absolute, X
1E	ASL absolute, X
1F	Future expansion
20	JSR absolute
21	AND (zero page, X)
22	Future expansion
23	Future expansion
24	BIT zero page
25	AND zero page
26	ROL zero page
27	Future expansion
28	PLP implied
29	AND #immediate
2A	ROL accumulator
2B	Future expansion
2C	BIT absolute
2D	AND absolute
2E	ROL absolute
2F	Future expansion
30	BMI relative
31	AND (zero page), Y
32	Future expansion
33	Future expansion
34	Future expansion
35	AND zero page, X
36	ROL zero page, X
37	Future expansion

38	SEC implied	60	RTS implied
39	AND absolute, Y	61	ADC (zero page, X)
3A	Future expansion	62	Future expansion
3B	Future expansion	63	Future expansion
3C	Future expansion	64	Future expansion
3D	AND absolute, X	65	ADC zero page
3E	ROL absolute, X	66	ROR zero page
3F	Future expansion	67	Future expansion
40	RTI implied	68	PLA implied
41	EOR (zero page, X)	69	ADC #immediate
42	Future expansion	6A	ROR accumulator
43	Future expansion	6B	Future expansion
44	Future expansion	6C	JMP (indirect)
45	EOR zero page	6D	ADC absolute
46	LSR zero page	6E	ROR absolute
47	Future expansion	6F	Future expansion
48	PHA implied	70	BVS relative
49	EOR #immediate	71	ADC (zero page), Y
4A	LSR accumulator	72	Future expansion
4B	Future expansion	73	Future expansion
4C	JMP absolute	74	Future expansion
4D	EOR absolute	75	ADC zero page, X
4E	LSR absolute	76	ROR zero page, X
4F	Future expansion	77	Future expansion
50	BVC relative	78	SEI implied
51	EOR (zero page), Y	79	ADC absolute, Y
52	Future expansion	7A	Future expansion
53	Future expansion	7B	Future expansion
54	Future expansion	7C	Future expansion
55	EOR zero page, X	7D	ADC absolute, X
56	LSR zero page, X	7E	ROR absolute, X
57	Future expansion	7F	Future expansion
58	CLI implied	80	Future expansion
59	EOR absolute, Y	81	STA (zero page, X)
5A	Future expansion	82	Future expansion
5B	Future expansion	83	Future expansion
5C	Future expansion	84	STY zero page
5D	EOR absolute, X	85	STA zero page
5E	LSR absolute, X	86	STX zero page
5F	Future expansion	87	Future expansion

88	DEY implied
89	Future expansion
8A	TXA implied
8B	Future expansion
8C	STY absolute
8D	STA absolute
8E	STX absolute
8F	Future expansion
90	BCC relative
91	STA (zero page), Y
92	Future expansion
93	Future expansion
94	STY zero page, X
95	STA zero page, X
96	STX zero page, Y
97	Future expansion
98	TYA implied
99	STA absolute, Y
9A	TXS implied
9B	Future expansion
9C	Future expansion
9D	STA absolute, X
9E	Future expansion
9F	Future expansion
A0	LDY #immediate
A1	LDA (zero page, X)
A2	LDX #immediate
A3	Future expansion
A4	LDY zero page
A5	LDA zero page
A6	LDX zero page
A7	Future expansion
A8	TAY implied
A9	LDA #immediate
AA	TAX implied
AB	Future expansion
AC	LDY absolute
AD	LDA absolute
AE	LDX absolute
AF	Future expansion
B0	BCS relative
B1	LDA (zero page), Y
B2	Future expansion
B3	Future expansion
B4	LDY zero page, X
B5	LDA zero page, X
B6	LDX zero page, Y
B7	Future expansion
B8	CLV implied
B9	LDA absolute, Y
BA	TSX implied
BB	Future expansion
BC	LDY absolute, X
BD	LDA absolute, X
BE	LDX absolute, Y
BF	Future expansion
C0	CPY #immediate
C1	CMP (zero page, X)
C2	Future expansion
C3	Future expansion
C4	CPY zero page
C5	CMP zero page
C6	DEC zero page
C7	Future expansion
C8	INY implied
C9	CMP #immediate
CA	DEX implied
CB	Future expansion
CC	CPY absolute
CD	CMP absolute
CE	DEC absolute
CF	Future expansion
D0	BNE relative
D1	CMP (zero page), Y
D2	Future expansion
D3	Future expansion
D4	Future expansion
D5	CMP zero page, X
D6	DEC zero page, X
D7	Future expansion

D8	CLD implied
D9	CMP absolute, Y
DA	Future expansion
DB	Future expansion
DC	Future expansion
DD	CMP absolute, X
DE	DEC absolute, X
DF	Future expansion
EØ	CPX #immediate
E1	SBC (zero page, X)
E2	Future expansion
E3	Future expansion
E4	CPX zero page
E5	SBC zero page
E6	INC zero page
E7	Future expansion
E8	INX implied
E9	SBC #immediate
EA	NOP implied
EB	Future expansion
EC	CPX absolute
ED	SBC absolute
EE	INC absolute
EF	Future expansion
FØ	BEQ relative
F1	SBC (zero page), Y
F2	Future expansion
F3	Future expansion
F4	Future expansion
F5	SBC zero page, X
F6	INC zero page, X
F7	Future expansion
F8	SED implied
F9	SBC absolute, Y
FA	Future expansion
FB	Future expansion
FC	Future expansion
FD	SBC absolute, X
FE	INC absolute, X
FF	Future expansion

6 The Oric's Memory Map



General Index

If you are no longer boggled by BASIC, try being mesmerized by machine code.

Talk to your computer in its own language and see how much more quickly it responds. Find out how it uses:

- hexadecimal and binary
- registers
- absolute and indirect addressing
- flags
- shifts, rotates and jumps

This self-contained book gives a full description of all the machine code instructions available to the Oric's 6502 chip and also suggests applications for their use. It describes how, when and where machine code routines can be entered. A simple monitor program is provided to facilitate the entry of your machine code routines.

Don't let your Oric gather dust on the shelf – talk to it!

This volume is suitable for both the Oric-1 and the new Atmos.

GB £ NET +006.95

ISBN 0-906812-69-0

00695

9 780906 812693

Shiva Publishing Limited